人気ナンバー1! **Java入門書の決定版**

大手ネット書店Java部門年間ランキング**第1位** ベストセラー 獲得

オブジェクト指向で挫折しない!「どうしてそうなるの?」

も
提供中!

インプレス

本書をスムーズに読み進めるためのコツ！

・PC でもスマホでも、ブラウザ上で Java プログラミング体験ができる**クラウド開発実行環境「dokojava」**を活用すれば、開発環境の準備でつまずくことなく、場所を選ばずに学習できるので、今すぐ Java プログラマーへの一歩を踏み出せます（詳細は p.4 参照）。

・「本格的な Java プログラマーを目指したいので、JDK のセットアップなど開発環境の準備方法も知りたい！」という方も安心。**付録 A「JDK を用いた開発」**（p.209）で開発環境の準備について解説していますので参考にしてください。

・「ちゃんと打ち込んでいるのにうまくいかない」「なぜか警告が出る」などの問題が起きましたら、陥りやすいエラーや落とし穴をまとめた**巻末付録 C「エラー解決・虎の巻」**（p.627）をご確認いただくと、解決できる場合があります。

※本書掲載の主要なソースコード一式をダウンロードできるようになりました（2015 年 3 月）。
http://book.impress.co.jp/books/1113101090

「dokojava」は株式会社フレアリンクが提供するサービスです。「dokojava」に関するご質問につきましては、株式会社フレアリンクへお問い合わせください。

インプレスの書籍ホームページ

書籍の新刊や正誤表など最新情報を随時更新しております。

http://book.impress.co.jp/

まえがき

著者の2人は、新入社員をはじめ若手エンジニアの方々の「Java 学習」を、これまで数多くお手伝いさせていただきました。「楽しく、しかし現場で本当に求められるスキルを」と考え実践してきた以下の3つのコンセプトに沿って、本書も執筆しています。

1. 手軽に・つまずかずに、Java をはじめられる

Java 特有の「複雑な開発準備作業」でつまずくことなく、最初の一歩を今すぐ踏み出せるよう、クラウド開発実行環境「dokojava」を準備しました。また、陥りがちなトラブルへの対策を巻末付録Cにまとめましたのでご参照ください。

2. 実務で役立つ内容に絞って学習を進めることができる

ネット活用の日常化を鑑み、「必要になったら自力で言語仕様を調べればわかる部分」の取り扱い優先度を下げ、実務に必要となる内容を重視しました。

3.「オブジェクト指向」の本質とおもしろさが理解できる

Java 学習の要である「オブジェクト指向」をスッキリ理解するために必要なのは文法知識ではなく、その根底を流れる思想・概念・用途のイメージです。Java 言語仕様に記載がないこの部分こそ、解説書がていねいに伝えるべき内容だと考え、ページ数を割きました。

この3つの硬派なコンセプトを、親しみやすいイラストとゲームという題材によってやわらかく包み仕上げたものが本書です。

この第2版では、最新 **Java8** への対応や **Eclipse** に関する追補に加え、より快適に読み進めていただけるよう 2,000 か所以上のアップデートを行いました。

「楽しく読み進めていったら、いつのまにか実務に耐えうるスキルが身に付いていた。」——そんな体験を、この本を手に取ってくださったみなさんにお届けできれば光栄です。

著者

【謝辞】
イラストの高田様ほか、デザイナー、編集の方々、教え方を教えてくれた教え子のみなさん、応援してくれた家族、この本に直接的・間接的に関わったすべての皆様に心より感謝申し上げます。

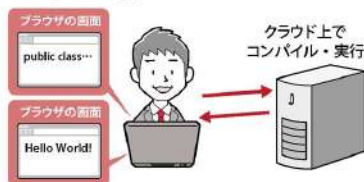
「dokojava」の使い方

1 「dokojava」とは

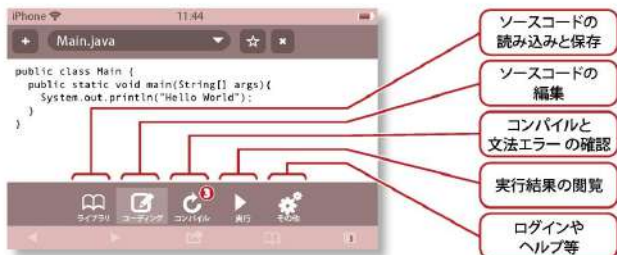
「dokojava」(どこじゃば)とは、パソコンやスマートフォンのブラウザだけでJava の開発と実行を行えるクラウドサービスです。dokojava を使えば、めんどうな「PC への開発環境の構築」をすることなく、PC や携帯端末から以下の URL にアクセスするだけで、今すぐ Java プログラミングを体験できます。

http://dokojava.jp

●PC 版　対応ブラウザ
IE, Firefox, Safari, Chrome, Opera

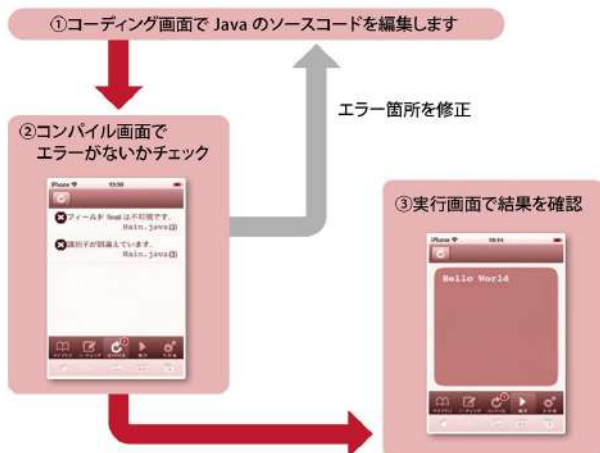
●携帯端末版　対応ブラウザ
Android および iPhone、iPad、iPod touch の Safari

※「dokojava」は株式会社フレアリンクが提供するサービスです。「dokojava」に関するご質問につきましては、株式会社フレアリンクへお問い合わせください。

2 「dokojava」の画面

dokojava は、5 つの画面を切り替えながら利用します。

画面写真は携帯端末版のものですが、PC 版も機能はほぼ同様です。また、画面デザインと機能は改良のため予告なく変更されることがあります。

3 ユーザー登録

dokojava での開発には、ユーザー登録（無料）とログインが必要です。「その他」画面から登録とログインを行ってください。

4 開発の流れ

5 「ライブラリ」の利用

本書掲載のソースコードは「ライブラリ」画面から読み込むことが可能です（リスト番号がないものについては読み込めないことがあります）。また、自分が開発したソースコードを「ライブラリ」に保存して後から読み込むことも可能です。

6 困ったときは

詳細な利用方法を知りたいときや困ったときは、ヘルプを参照してください。

http://dokojava.jp/help

CONTENTS

CONTENTS

第Ⅱ部 すっきり納得 オブジェクト指向

CONTENTS

第Ⅲ部　もっと便利に API 活用術

COLUMN

本書の見方

本書には、理解の助けとなるさまざまな用意があります。押さえるべき重要なポイントや覚えておくと便利なトピックなどを、要所要所に楽しいデザインで盛り込みました。読み進める際にぜひ活用してください。

本文中の色文字：
本文中、重要な用語や特に注意していただきたい部分に色を付けました。

アイコン：
各アイコンの示す内容についてはこのページの下「アイコンの種類」でご確認ください。

吹き出し会話：
本書にはみなさんと一緒に Java を学ぶ仲間たちが登場します（p.15 参照）。彼らが繰り広げる会話の中にも、学びの場や開発現場でありがちな疑問点やひらめきが詰め込まれています。実は最も重要なポイントが含まれていることも…。ぜひお見逃しなく！

予約語：
この色文字は**予約語**です（p.46 参照）。

コメント：
この部分は**コメント**です（p.41 参照）。

注目コード：
解説をスムーズに理解するため注目すべき部分です。

各章の練習問題：
各章の章末には練習問題が付いています。ここでその章の理解度を確認します。あまりできていない場合は、もう一度その章を読み返してみましょう。

各章のまとめ：
その章で学んだことをまとめています。それぞれ頭に入っているか、正しく理解できているかを思い起こし、達成度を測るチェック表としてお使いください。

アイコンの種類

ポイント紹介：
本文における解説で、特に重要なポイントをまとめています。

構文紹介：
Java で定められている構文の記述ルールです。正確に覚えるようにしましょう。

文法上の留意点：
構文を記述するときの文法上の注意点などを紹介します。

コラム：
本編では詳細に取り上げないものの、知っておくと重宝する補足知識やトリビアなどを紹介します。

第0章

Javaを はじめよう

この本を手に取ってくださったみなさんは、
「Java に興味を持つ」という貴重な一歩をすでに踏み出しています。
そのかけがえのない一歩目を大切にしながら、
つまずくことなく二歩目・三歩目を踏み出していきましょう。

CONTENTS

0.1 ようこそ Java の世界へ

0.1.1 Java を使ってできること

Java とはプログラムを作るために利用するプログラミング言語の 1 つです。Java を使えば、さまざまなコンピュータで動作する多様なプログラムを開発することができます。

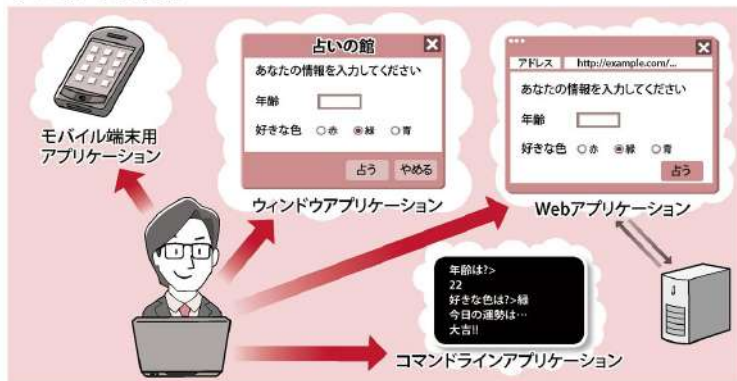



図 0-1　Java を使ってできること

次のような特徴から、Java はさまざまな分野で利用されています。

- 学びやすく標準的な基本文法
- 大規模開発を支援するオブジェクト指向に対応
- 豊富に準備された便利な命令群
- 多様なコンピュータで同じように動作する汎用性

このような特徴を持つ Java を自由自在に操り、プログラムを組めるようになりたいと思う人のために、この本は生まれました。初めてプログラミング言語に触れるという方にも「Java の基本文法」から「オブジェクト指向」に至るまで、スッキリ理解できて楽しく読み進められるよう心がけました。ぜひ実際に手を動かしてプログラミングをしながら、一緒に Java をマスターしていきましょう。

0.1.2 一緒に Java を学ぶ仲間たち

この本でみなさんと一緒に Java を学んでいく 3 人を紹介しましょう。

図 0-2　一緒に Java を学ぶ仲間たち

0.2 はじめてのプログラミング

準備はいいかな？　さっそく Java プログラミングを体験してみよう。

はい！

0.2.1 プログラミングの準備をしよう

でも… Java のプログラムを作るためには高いソフトや最新のパソコンが要るんじゃないですか？

私も難しい設定って、したことなくて…。

いや、普通のパソコンがあれば今すぐ作れるよ。

Java プログラミングを始めるために高価なソフトや特別な機材は必要ありません。しかし、いくつかのソフトのインストールや少し高度なパソコンの設定変更といった準備作業が必要になります。

実は、その準備作業でつまずく人も少なくありません。そこで本書では、インターネットにつながるパソコン（Windows、Mac）やスマートフォンがあれば今すぐ Java プログラミングを体験できるしくみを用意しました。それが「どこでもクラウド Java 開発実行環境」略して「dokojava」です。Web ブラウザを起動して以下のアドレスにアクセスしてください（p.4 も参照してください）。

http://dokojava.jp

図0-3　dokojava（iPhone版）の画面

図 0-3 は iPhone の Web ブラウザで、dokojava にアクセスした画面です。多少デザインが異なりますが、パソコン版も機能は同じです（詳細な機能・操作方法などは、dokojava のヘルプを参照してください）。

dokojava では、次の 3 つの手順で Java プログラミングをします。

① プログラムの入力

画面に Java のプログラム（本書のサンプルプログラム）を入力していきます。

② コンパイル

dokojava が入力したプログラムを検査し、実行の準備をします。

③ 実行

プログラムを実行し、結果が画面に表示されます。

この 3 ステップはプログラミングの基本手順なんだ。詳しいことは後でまた解説するよ。

dokojava が利用できないときは

dokojava はさまざまな端末から利用できるように作られていますが、うまく利用できない場合は　http://dokojava.jp/help　にアクセスして解決のヒントを探しましょう。また、メンテナンスでサービスが停止中の場合は、しばらく時間をあけて再度アクセスしてみてください。

0.2.2 サンプルプログラムを動かしてみよう

それではまず、サンプルプログラムを動かしてみよう。

初めての Java プログラミング… どきどきしますね。

dokojava にアクセスすると、画面にはすでに次のようなサンプルプログラムが入力されているはずです（左端の数字はリストの行番号を示します。これはプログラムではないため入力しません）。

リスト 0-1　HelloWorld プログラム

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("Hello World");
4    }
5  }
```

このプログラムは画面に「Hello World」という文字を出すという単純なものですが、現時点でそのしくみを理解する必要はありません。dokojava の「コンパイル」ボタンをクリックし、さらに「実行」ボタンを押せば次のような実行結果が表示されるでしょう。

```
Hello World
```

0.2.3 画面に好きな文字を表示させよう

おおっ…。このプログラムでこの表示が出るってことは…
ひょっとしてここを書き換えれば…。

それではプログラムを少し書き換えて、自分の思ったとおりの文字を画面に表

示させてみましょう。先ほどのプログラムの中の「Hello World」の部分を書き換えてください。英文はもちろん日本語でも大丈夫です。書き換え終わったら、コンパイルボタンと実行ボタンを押します。

リスト 0-2

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     System.out.println("すがわら");
4   }
5 }
```

この部分を書き換えた

実行結果

すがわら

よし、さらに書き換えて、コンパイル・実行っと…あれ？ おかしいなぁ…エラーが表示されちゃいます。

プログラムに誤りがあるとコンパイルエラーが報告されます。誤りを取り除かない限り、コンパイルは完了せず、実行もできません。さて、湊くんが作った次のプログラムのどこに誤りがあるかわかりますか？

リスト 0-3

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     System.out.println("湊くんかっこいい！最高！);
4   }
5 }
```

湊くんは、かっこ良くも最高でもないから…。

そこは関係ないだろ！「最高！」の後に " 記号がないからかな？

湊くんの言うとおり文字列の後に " 記号がないためエラーが発生しました。プログラムを修正して再びコンパイル・実行します。

実行結果

```
湊くんかっこいい！最高！
```

0.2.4 たくさんの文章を表示しよう

画面に 2 行以上の文章を表示させることもできます。次のように 2 行を書き足してコンパイル・実行してみましょう。

リスト 0-4

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("すがわら");
4      System.out.println("31歳です");
5      System.out.println("お酒が好きです");
6    }
7  }
```

この 2 行を追加した

実行結果

```
すがわら
31歳です
お酒が好きです
```

0.2.5 計算させてみよう

いい調子だね。では、さらに「プログラムらしい」ことにチャレンジしよう。

では、コンピュータに計算をさせてみましょう。次のように2行を書き足して、コンパイルと実行をします。

リスト 0-5

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     System.out.println("すがわら");
4     System.out.println("31歳です");
5     System.out.println("お酒が好きです");
6     System.out.println("31 + 31の計算をします");
7     System.out.println(31 + 31);
8   }
9 }
```

数式

この2行を追加した

実行結果

```
すがわら
31歳です
お酒が好きです
31 + 31の計算をします
62
```

なるほど！　数式を書いたら、その計算をしてくれるんですね。

もっと数式を複雑にしてみるのもいいでしょう。ぜひリスト0-5の7行目と8行目の間に以下の行を追加して動作を確かめてみてください。

```
System.out.println(35 - 10);        → 25
System.out.println(-5 * 2);         → -10
System.out.println(6 * 6 * 3.14);   → 113.04
System.out.println("こたえは" + 64); → こたえは 64
```

0.2.6 変数を使ってみよう

数学ではxやyといった文字を数式に使いましたね。Javaでも似たようなことができます。まだ解説していない記述も出てきますが、見よう見まねでリスト0-5に次の3行(8〜10行目)を追加して、コンパイル・実行してください。

リスト 0-6

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("すがわら");
4      System.out.println("31歳です");
```

```
5       System.out.println("お酒が好きです");
6       System.out.println("31 + 31の計算をします");
7       System.out.println(31 + 31);
8       int x;      変数xを準備する
9       x = 6;      xに6を入れる
10      System.out.println(x * x * 3.14);
11    }
12  }
```

実行結果

```
すがわら
31歳です
お酒が好きです
31 + 31の計算をします
62
113.04      変数を使った計算結果
```

0.2.7 プログラミング体験を終えて

なんだか、Javaを使えばいろんなプログラムを作れる気がしてきました。

ボクは子どもの頃からずっとゲームを作りたかったんです。いつかJavaでRPGを作りたいなぁ…。

えぇ～！ もっとまじめに「ネットショップ」だとか「金融システム」とかそういうの作りたいと思わないの？

なるほど、ゲームかぁ… 学習の題材としては悪くないなぁ…。

初めてのプログラミング体験はここまでですが、もっと複雑で高度なプログラムを作ることも可能です。実際世の中では、Javaで開発されたネットショップ、金融システム、そしてRPGが動いています。おおよそみなさんが思いつくプログラムの多くがJavaで開発されているか、開発することが可能なものでしょう。

ぜひあなたも「いつか作ってみたいプログラム」を自由に想像してみてください。

図0-4　この本による解説の全体像

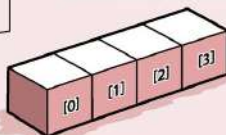
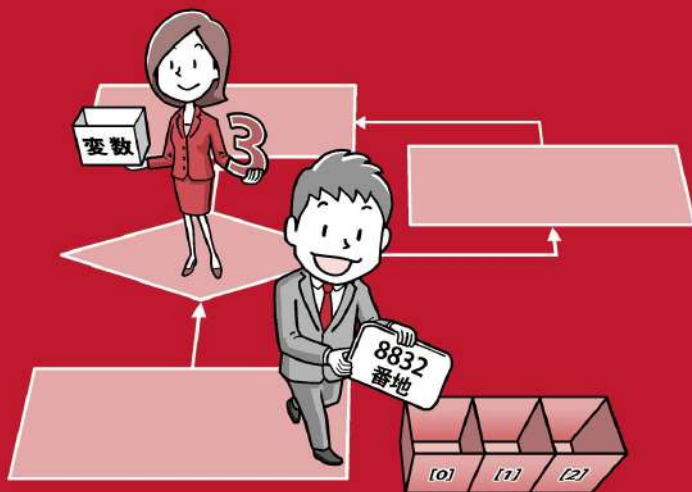
Start!

いろいろな分野を巡りながら
Javaを学んでいこう!

第II部 オブジェクト指向

効率的に本格的プログラムを
作るための考え方を学びます

さて、いよいよ次節からはJavaの学習に入っていきます。湊くんが夢見るRPGには、さまざまなJava学習のエッセンスが詰まっていますので、本書の前半で基本文法をしっかり学習した上で、後半ではRPG開発を題材に楽しく学習を進めていきましょう。

第 I 部

ようこそ Javaの世界へ

Javaプログラミングことはじめ

まだ1時間も経っていないのに、Javaのプログラムを作って動かせちゃったね♪

そうだね。あ、菅原さん。Javaにはスゴい命令とか便利な機能がいっぱいあるんですよね？！　RPGのアイデアがいっぱいわいてきちゃって、ボク、一日も早くJavaをマスターしたいんです！

まぁあせらずに。せっかくだから、ちゃんと腰を据えて学ぼうじゃないか。しっかり基礎を固めれば、立派なRPGが作れるようになるよ。

まずは何から学んだらいいのかしら。

基本的な文法と命令を理解して、指示するとおりにJavaを操れるようになることから始めよう。学ぶことは多いけど、ていねいに1つずつ、着実に理解していけば必ずマスターできるよ。

はい、よろしくお願いします！

みなさんは、第0章を通して、コンピュータにさせたい処理を命令として書いておけば、そのとおりにJavaが動いてくれることを体験できたと思います。Javaが定めるさまざまな命令や文法を理解し、使いこなせば、とても複雑な処理をコンピュータにさせることも可能です。
第I部では、これらJavaが定める基本的な文法について紹介します。この部の6つの章を学び終えれば、コンピュータにひととおり指示を与えられるようになるはずです。

第 1 章

プログラムの書き方

Java では、プログラムの書き方に関するさまざまなルールが定められています。
なかでも、作るプログラムの内容や規模に関係なく、必ず使うことになる基本的なルールはとても重要です。
まずはそれら基本的な文法をしっかりおさえることから学習を始めましょう。

CONTENTS

1.1 Java開発の基礎知識

1.1.1 開発の流れ

プログラミング体験で繰り返し行ったように、Javaの開発は、①ソースコードの作成、②コンパイル、③実行、この3ステップで行います（図1-1）。では、それぞれのステップを詳しく解説していきましょう。

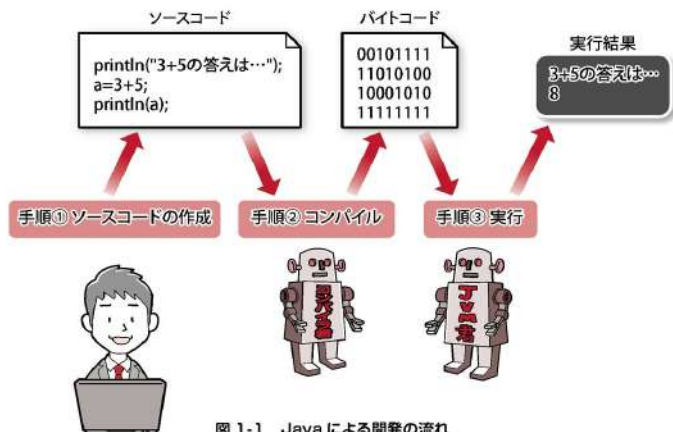



図1-1　Javaによる開発の流れ

手順①　ソースコードの作成

最初にJavaが定める文法に従ってコンピュータへの命令を記述していきます。「public class ～」のような記述で、人が読める状態のプログラムをソースコード（source code）、または単にソースと言います。

手順②　コンパイル

コンピュータは、その心臓部であるCPUがプログラムを解釈しながら指示どおりに動きます。しかし、CPUは私たちが書いたソースコードを直接、読んで動くことはできません。CPUはマシン語（machine code）と呼ばれるCPUが理解できる言葉で書かれたプログラムしか実行できないのです。

そこで私たちは、まずソースコードに対してコンパイルという処理を行ってバイトコード（byte code）という状態に変換します。この際、ソースコードの文法チェックも行われ、もし誤りがあればコンパイルは失敗し、誤りの箇所が表示されます。この一連の処理はコンパイラ（compiler）と呼ばれる変換ソフトウェアが担当します。

手順③　実行

コンパイルが無事完了したら、インタプリタ（interpreter）と呼ばれるソフトウェアに対してバイトコードの実行を指示します。インタプリタはJVM（Java Virtual Machine）というしくみを内部に持っており、バイトコードを少しずつ読み込みながら、それをマシン語に変換してコンピュータのCPUに送ります。こうしてコンピュータはソースコードで指示したとおりに動作するのです。

1.1.2 開発環境の整備

Java プログラミングを行うために、私たち開発者はコンパイラとインタプリタを準備する必要があります。その具体的な方法は次の２つです。

①自分の PC にコンパイラとインタプリタをインストールする

インターネットから Java のコンパイラやインタプリタをダウンロードして、それを自分の PC にインストール（導入）します。Java 開発者の多くは、この方法を用いますが、初心者には少し難しい手順が含まれています。

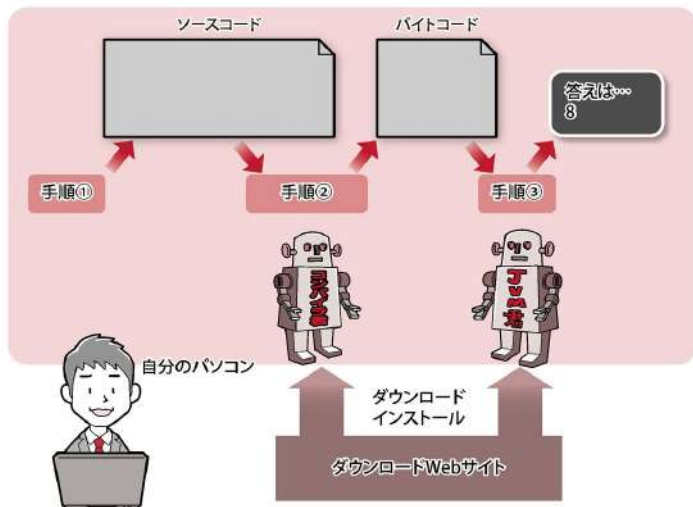



図 1-2　自分の PC に構築した開発環境による開発

② dokojava サーバにコンパイルと実行をさせる

インターネット上に準備されている dokojava サーバには、コンパイラとインタプリタがすでに導入されています。開発者がブラウザ画面からソースコードを送れば、サーバ上でコンパイルと実行が行われ、その結果が Web ブラウザの画面に戻されます。

難しいセットアップなしでJavaのプログラミングが可能なメリットがありますが、さまざまな理由からJavaの機能を一部、利用できない制約があります。

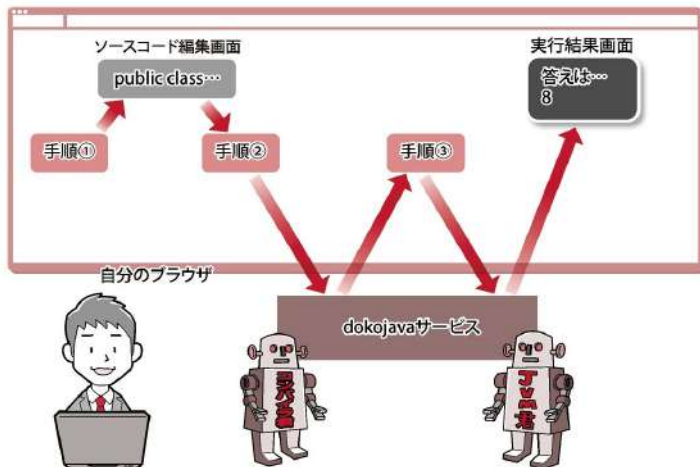



図 1-3　dokojava を用いた開発

この本で取り扱う範囲のプログラムを学習用に作成して動かす目的であれば、どちらの方法でも構いません。dokojavaを利用して学習を進める方は、このまま次節以降に進んでください。

自分のPCにコンパイラとインタプリタを導入して学習を進めたい方は、先に付録A「JDKを用いた開発」を参考にインストールを行ってください。

開発の流れがわかり、開発環境も揃いました。
あとはソースコードを書くだけですね！

次節からはソースコードの記述方法を解説していくよ。

Javaプログラムの基本構造

1.2.1 プログラムの骨格

プログラミングを体験して感じましたけど、いろいろ命令を書いていくのは「真ん中の部分」だけなんですね。

よく気づいたね。Javaのプログラムには基本的な構造があるんだよ。

まずJavaプログラムの全体像を見てみましょう。

リスト1-1

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     System.out.println("RPG:スッキリ魔王征伐");
4     System.out.println("Ver.0.1 by 湊");
5     System.out.println("<ただいま鋭意学習・制作中>");
6     System.out.println("プログラムを終了します");
7   }
8 }
```

処理・命令の部分（3〜6行目）

Javaのソースコードには、波カッコ{…}で囲まれた部分が多く登場します。この波カッコで囲まれた部分を**ブロック**（block）と呼びます。外側のブロックはクラスブロック（class block）、内側のブロックはメソッドブロック（method block）と呼ばれ、Javaのソースコードは必ずこれらのブロックによる二重構造を持っています。

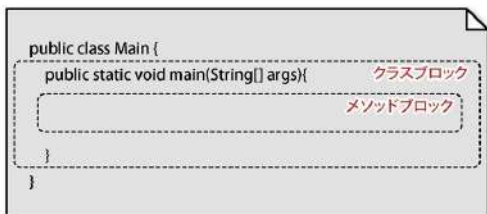



図 1-4　Java のソースコードのブロック構造

私たちが「コンピュータに対する指示・命令」を記述していくのはメソッドブロックの中です。それより外側（最初の2行と最後の2行）は、どのようなプログラムを開発する場合も、あまり変わらない「お決まりのパターン」です。

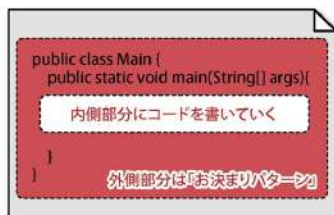



図 1-5　外側の部分はお決まりのパターン

お手紙を書くときに、本文の前後に「時候の挨拶」と「結語」を書くのと似ていますね。

外側部分は毎回ほとんど同じだから、何度も書いて覚えてしまおう。

ソースコードの外側部分はお決まりパターンとはいえ、どんなプログラムを開発するときも「毎回まったく同じ記述でいい」わけではありません。1行目にある「public class」の直後に書かれる単語は、このプログラムの名前を指定するものです。正式には**クラス名**（class name）といい、大文字のアルファベットで始まる名前を付けることが一般的です。

```
public class MyDiary {
  public static void main(String[] args) {
    ⋮
```

クラス名の指定

クラス名はとても大事です。なぜなら Java には次のルールがあるからです。

Javaソースファイルの名前

Java のソースコードを記述したファイル（ソースファイル）を保存するときには、ファイルの名前は「クラス名 .java」にしなければならない。

Java プログラムのクラス名に対するルールを次にまとめました。

MyDiary.java

```
public class MyDiary {
  public static void main(String[] args) {
    ⋮
  }
}
```

クラス名の指定

※ソースファイル名は「クラス名 .java」にする。
※クラス名はアルファベット大文字で始める。

1.2.2 プログラムの書き始め方

1.2.1 項で学んだことをまとめると、私たちは次の流れでソースコードを作っていくことになります（図 1-6）。

Java プログラムの書き始め方

①どのようなプログラムを作りたいかを考えます。
②プログラムの名前を決めます（クラス名が決まります）。

③「クラス名.java」という名前でファイルを作ります。
④ソースコードの外側部分を記述します。
⑤ソースコードの内側部分に命令を書いていきます。

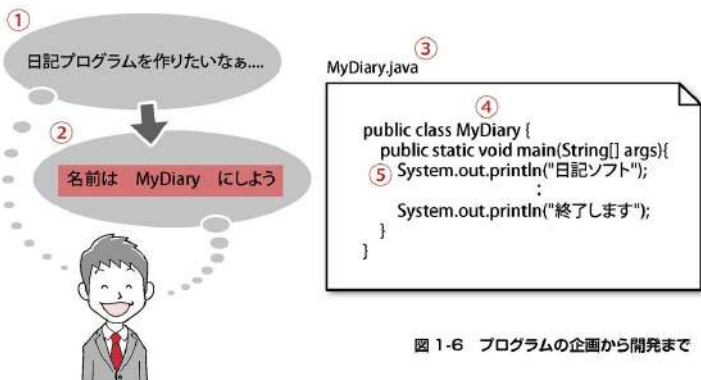



図 1-6 プログラムの企画から開発まで

ソースコードの記述に際しては、意識すべき大切なことが3つあります。

大切な意識 1:正確に記述する

ソースコードには、さまざまな文字・数字・記号が登場します。見た目が似ていても、間違った文字を入力するとプログラムは正常に動きません。特に次の点に気をつけましょう。

- **英数字は基本的に半角で入力し、大文字小文字の違いも意識する。**
- **オー(o・O)とゼロ(0)、エル(l)と数字のイチ(1)、セミコロン(;)とコロン(:)、ピリオド(.)とカンマ(,)を間違えない。**
- **カッコ((), {}, [])や引用符(', ")の種類を間違えない。**

特に正確な記述を求められるのはプログラムの2行目です。「public static void main (String[] args)」を一字一句間違えずにスラスラ書けるようになりましょう。

public static void…、うーん覚えられるかなぁ?

リズム良く口に出すと覚えやすいよ。
「パブリック・スタティック・ボイド・メイン。ストリング・カッコカッコ・エーアールジーエス」ってね。

ちょっと恥ずかしいけど、覚えるまでは入力しながら声に出してみます。

大切な意識 2:「上から下へ」ではなく「外から内へ」

プログラミングを始めてしばらくは、「ブロックの"{"と"}"の対応が正しくない」というエラーに悩まされがちです。特にブロックの閉じ忘れ("}"の書き忘れ)に悩まされる初心者に共通するのは、「ソースコードを上から順番に記述している」ことです。

ソースを上から1行ずつ記述すると「①ブロックを開く→②中身を書く→③ブロックを閉じる」という手順になるので、中身を書いている間に「ブロックを閉じる必要があること」を忘れてしまいます。

図 1-7　ソースコードを上から書いていくと、ブロックを閉じ忘れやすい

このエラーを防ぐため、次の図のように「ブロックを**開いてすぐに閉じる**」「中身を一生懸命作る」という手順で書くようにするとよいでしょう（メソッドについては1.2.5項で解説します）。

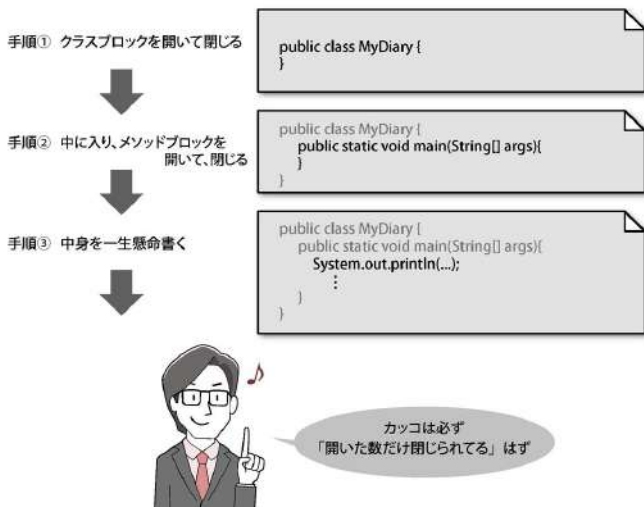



図1-8　ソースコードを外から内へ書いていくと、閉じ忘れの失敗が減る

大切な意識3:読みやすいコードを記述する

文法的には誤りがなくても、「人間が読みにくい煩雑なコード」や「複雑すぎて内容の理解に時間がかかるコード」では修正や改良が難しくなります。特に仕事でJavaプログラムを作る人は、同僚や取引先にソースコードを見てもらうことがあるため、誰が見てもわかりやすい記述をするようにしましょう。

先輩。具体的にどういう工夫をしたら「読みやすいコード」が書けるようになりますか？

「インデント」と「コメント」を上手に活用するといいよ。

1.2.3 インデント

Java では「ソースコード中、どこに改行や空白を入れるかは基本的に自由」とされています（ただし、public などの単語が途中で切れてしまうような改行や空白の挿入は許されません）。極端な例ですが、図 1-9 のようにまったく改行せずにソースコードを記述してもコンパイルは成功します。

```
public class Main {public static void main(String[] args) {System.out.println
("フリーフォーマットの実験");}}
```

図 1-9　改行せずに書いたソースコード。これでもプログラムは動くが…

しかし、これではプログラムの構造を把握することは難しいですね。そこで、適切な場所で改行や空白を入れるようにしましょう。特に**ブロックの開始と終了では正確に字下げを行い、カッコの対応とブロックの多重構造の見通しを良くする**ことがポイントです。この字下げのことを**インデント**（indent）と呼び、キーボードの TAB キーで行うことが一般的です（図 1-10）。

図 1-10　インデントを入れたソースコード。ブロックの構造がわかりやすくなる

インデントは、でたらめに入れてはいけません。誤ったインデントはプログラム構造の読み間違い、そして致命的なエラーの原因になることもあるからです。

図 1-11　混乱を招くインデント

とはいっても、ちょっとぐらいインデントが変でも、まぁ動くし…いいじゃないですか？

ダメだよ。以降の章では、ブロックが4重や5重の入れ子構造になる書き方も登場するし、良くない書き方を最初に身に付けてしまうと、その後の学習効率にも大きく影響を与えてしまう。今のうちに「正確なインデント」を入れる習慣を身に付けてほしい。

1.2.4 コメント

今は10行程度だけど、大きいコードになると「何行目で何をやっているか」わからなくなってしまいそうですね。

大丈夫。ソースコードの中には解説文を書き込めるんだ。もちろん日本語もOKだよ。

よりプログラムを読みやすくするため、ソースコード中に解説文を書き込むこともできます（次ページ図1-12）。この解説文を**コメント**（comment）といい、プログラムのコンパイル時と実行時には無視されます。人が読むためだけに書かれるものですから日本語で書いて構いません。

コメント文①　複数行コメント

/* コメント本文（複数行でも可）*/

コメント文②　単一行コメント

// コメント本文（行末まで）

```
/* サンプルプログラム Main
   開発者: 菅原        作成日: 2011年4月
*/
public class Main {

    // ここからmainメソッド
    public static void main(String[] args) {
        int age;  // 年齢を入れる箱
        age = 20;
        System.out.println("私は" + age + "歳");
    }
}
```

/* から */までは コメント

// から 行末までは コメント

図 1-12　コメントはソースコードのあらゆるところに記述できる

1.2.5 main メソッドの中身

残るは「ソースコードの内側部分」の記述方法だね。

どんなルールに従ってプログラムを書いていけばいいのかな？

1.2.1 項で学んだように、計算や表示など Java の命令を書いていく場所はメソッドブロックの内部です。ソースコードの 2 行目に「main」の表記が出てくることから、この部分は **main メソッド**とも呼ばれます（図 1-13）。

main メソッドの中には、**文**（statement）を順番に書いていきます。プログラムの実行時には、文は上から順に 1 行ずつ処理されていきます。

文の末尾には必ずセミコロン（;）を付けるのが Java のルールだ。忘れやすいから注意してほしい。

図 1-13 の MyDiary のソースコードの例にあるように、main メソッドの中にはさまざまな文を書くことができます。Java には、とても多くの種類の文が存在しますが、図 1-14 に示した 3 種類に分類して考えると理解しやすいでしょう。

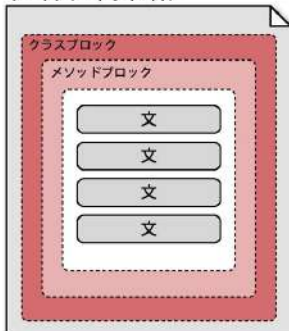
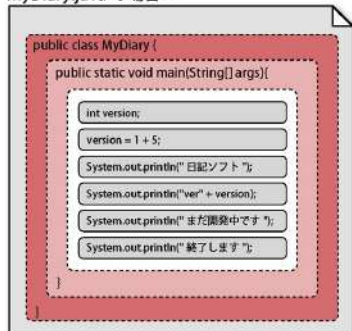



図 1-13　main メソッドのブロック内に「文」を書いた例

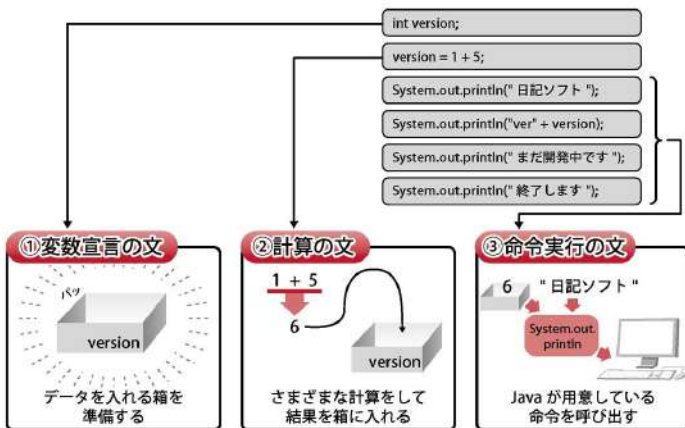



図 1-14　文は 3 種類に分けることができる

次節では、まず「①変数宣言の文」について学びましょう。

1.3 変数宣言の文

1.3.1 変数宣言の文とは？

変数宣言の文とは、「新たな変数を準備せよ」とコンピュータに指示する文です。**変数とはデータを格納するためにコンピュータ内部に準備する箱**のようなもので、数字や文字列などプログラムが扱うさまざまな情報を入れたり取り出したりできます。

変数の実体はコンピュータのメモリにある区画です。変数に値を入れると実際にはメモリに値が書き込まれます。では、実際に変数を利用している例を見てみましょう。

リスト 1-2　変数宣言の文

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int age;                    ← 変数宣言の文（ageという箱を用意）
4      age = 30;                   ← 箱に数字の「30」を入れる
5      System.out.println(age);    ← 箱の中身を表示
6    }
7  }
```

実行結果

```
30
```

このプログラムでは変数ageに値「30」を入れ、それを取り出して画面に表示しています。このように変数に値を入れることを「代入」、取り出すことを「取得」と呼びますが、どちらも変数を宣言した後にしかできません。

変数を使いたかったら、まずは「変数宣言の文」を使って宣言する必要があるんだね。

変数を宣言するときには、変数名（データを入れる箱の名前）と型（データを入れる箱の種類や大きさ）の２つを必ず指定する決まりになっています。

変数宣言の文

```
型　変数名;
```

型とは「変数に入れることができるデータの種類」のことです。たとえばリスト 1-2 で使われている int は「整数」を表す型です。この型で宣言された変数 age には整数しか入れることはできず、小数や文字列を入れることはできません。

なるほど！ ちなみに int のほかに、どんな型があるんですか？ 変数に付ける名前に決まりはあるんですか？　それと…。

まぁ落ち着いて。まずは「変数に付ける名前」について次項で説明しよう。

1.3.2　変数の名前

変数を宣言する際、私たちは変数に名前を付ける必要があります。Java プログラミングでは変数以外にも名前を付けることがありますが、それらの名前として使える文字や数字の並びのことを**識別子**（identifier）と呼びます。

名前を何にするかは基本的にプログラマの自由ですが、通常はアルファベット、数字、アンダースコア「_」、ドル記号「$」などを組み合わせて作ります。ひらがなや漢字を含めることも可能ですが、推奨されません。そのほかにも次ページのルールや慣習があるので注意してください。

■禁止されている単語を使ってはならない

Java では、「そもそも名前として使ってはいけない単語」とされている**予約語**（keyword）が約 50 個あります。これまでにも登場した int や void、public、static などは予約語ですので、これらを変数名として利用することはできません。

■すでに利用している変数名を再度使ってはならない

すでに変数 name を宣言しているのに、再び変数 name を宣言してはいけません。2 つの変数が区別できなくなってしまうからです。

■大文字・小文字・全角・半角の違いは区別される

大文字／小文字、全角／半角の違いは人間にはささいなものですが、Java では完全に区別されます。たとえば変数 name と変数 Name は別のものとして扱われますので注意してください。

■小文字で始まるわかりやすい名前を付けることが望ましい

慣習的に、変数には小文字で始まる名詞形の名前を付けます。また、複数の単語をつなげて変数名にする場合、2 つ目以降の単語の先頭は大文字にします。たとえば「myAge」などです。

また、格納される情報の内容を想像しやすく、わかりやすい変数名が望ましいでしょう。第 3 章で登場するループ変数など一部例外はありますが、a や s のような 1 文字の変数名は避けるようにしましょう。なお、本書では紙面スペースの都合により短い変数名を用いる場合があります。

基本的に自由でも、好き放題に名前を付けていいわけではないんですね。

Java の予約語一覧

以下は Java の予約語ですので、変数名には使えません。以降、本書のリストでは、予約語を「long」のように色付きで表します。

abstract、assert、boolean、break、byte、case、catch、char、class、const、continue、default、do、double、else、enum、extends、final、finally、float、for 、if、goto、implements、import、instanceof、int、interface、long、native、new、package、private、protected、public、return、short、static、strictfp、super、switch、synchronized、this、throw、throws、transient、try、void、volatile、while

1.3.3 データ型

変数宣言に使う「型」については、どんな種類があるんですか？いっぱいあったら覚えるのが大変だなぁ…。

まずは次の９つを覚えておけば大丈夫だよ。

プログラムで扱うことができるデータの種類のことを、**データ型**(data type)または単に**型**と言います。Java には多くの型が準備されていますが、今は次の９つだけを覚えておけばよいでしょう。

分類	型名	格納するデータ	変数宣言の例		利用頻度
整数	long	大きな整数	long worldPeople;	//世界の人口	△
	int	普通の整数	int salary;	//給与金額	◎
	short	小さな整数	short age;	//年齢	△
	byte	さらに小さな整数	byte glasses;	//所持する眼鏡の数	△
小数	double	普通の小数	double pi;	//円周率	○
	float	少しあいまいでもよい小数	float weight;	//体重	△
真偽値	boolean	trueかfalse	boolean isMarried;	//既婚か否か	○
文字	char	1つの文字	char initial;	//イニシャル1文字	△
文字列	String	文字の並び	String name;	//自分の名前	◎

図 1-15　代表的な９種類のデータ型

それでは、９つの型を４つのグループに分けて１つずつ紹介しましょう。

■整数を格納できる 4 つの型（long、int、short、byte）

long 型、int 型、short 型、byte 型の変数には整数を代入できます。

```
long worldPeople; worldPeople = 6900000000L;
int salary; salary = 152000;
short age; age = 18;
byte glasses; glasses = 2;
```

末尾の L は第 2 章で解説します

これら 4 つの型は箱の大きさ（コンピュータ内部で準備されるメモリの量）に違いがあります。そのため、代入できる値の範囲に図 1-16 の制限があります。

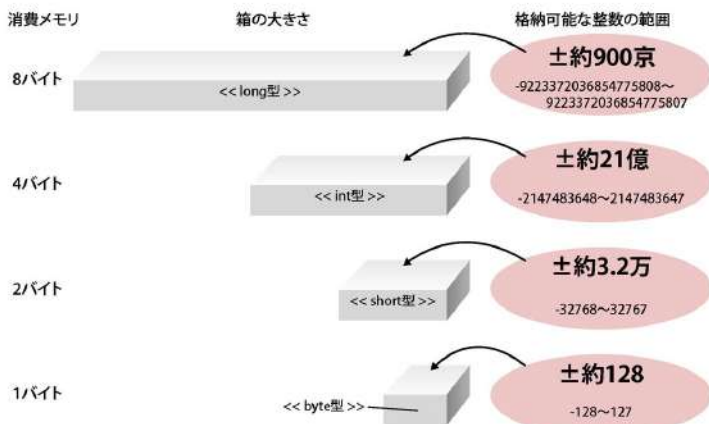



図 1-16　long 型、int 型、short 型、byte 型に代入できる整数の範囲

たとえば byte 型の変数を宣言した場合、消費するメモリは 1 バイトだけなので、これには－128 ～ 127 までの数字しか代入できません。

一方、long 型の変数を宣言した場合、8 バイトのメモリを消費しますが、－9223372036854775808 ～ 9223372036854775807 という、とても大きな整数を代入できます。

どれを使うのがベストか迷っちゃいそう。年齢なら byte 型で十分だし…いや、一応 short にしておいたほうが安全かしら…。

特殊な場合を除けば、常に int を使っておけば大丈夫だよ。

最近のコンピュータは多くのメモリを搭載しているため、これら 4 つの型を厳密に使い分ける必要があるケースはまれです。また、short や byte より int のほうが高速に処理できるコンピュータも多いので、**整数を代入したい場合、通常は int 型**を使えば問題ありません。

■小数を格納できる 2 つの型 (double, float)

double と float は「3.14」や「−15.2」といった小数を含む数値を代入するための型です。コンピュータの内部では小数を「浮動小数点」という形式で管理していることから**浮動小数点型**(floating point type)と総称されることもあります。具体的には以下のように用います。

```
double height; height = 171.2;
float weight; weight = 67.5F;
```

末尾の F は第 2 章で解説します

double のほうが float より多くのメモリを消費しますが、より厳密な計算を行うことができます。そのため、**特別な事情がない限り double 型を使用します**。

ここで 2 人に覚えておいてほしいことがあるんだ。忘れると大事故につながる大切なことだよ。

実は、浮動小数点方式には「**真に厳密な計算ができない**」という弱点があります。つまり、計算を行った際にわずかな誤差が発生することがあるのです。通常は無視できるほど小さな誤差ですが、それが積み重なると大きな問題になることもあります。そのため、誤差が許されない計算、特に**お金の計算に double や float を使ってはいけません**。

■ YESかNOかを格納できるboolean型

boolean型は、「YESかNOか」「本当か嘘か」「裏か表か」「成功か失敗か」といった二者択一の情報を代入するための型です。肯定的情報を意味する**true**、否定的情報を意味する**false**のどちらかの値のみを代入することができます。

```
boolean isMarried; isMarried = true;    意味:結婚している
boolean result; result = false;         意味:結果は失敗
```

なお、trueは真、falseは偽という意味を持つので、boolean型のことを**真偽値型**と言う場合もあります。

■ 1文字だけを格納できるchar型、文字列を格納できるString型

char型は全角・半角を問わず「1文字」を代入できる型です。一方、String型は文字列(0文字以上の文字の集まり)を代入できる型です。具体的には次のように使います。

```
char gender; gender = '男';
String name; name = "すがわら";
```

この例にあるように、ソースコードに「文字」データを記述する場合は引用符(')で囲みます。そして「文字列」データを記述する場合は二重引用符(")を使います。このため「char gender; gender = " 男 ";」とするとコンパイルエラーになります。

また、全角の引用符を使わないように気をつけよう。日本語の入力をした後で、ついつい後ろの「'」や「"」も全角にしてしまうミスがよくあるよ。

1.3.4 変数の初期化

何のために変数宣言をするかといえば、「変数にデータを入れたいから」です。

```
int age;
age = 22;
```

変数宣言の文

ageに「22」を代入

この文は次のように1行にまとめて書くこともできます。

```
int age = 22;
```

変数宣言と代入を1行で行う

このように「変数を宣言すると同時に値を代入すること」を**変数の初期化**と呼びます。

変数の初期化

型 変数名 = 代入するデータ;

1.3.5 定数の利用

変数には異なる値を何度でも代入できます。変数に値を入れた後、別の値を代入するとどうなるかをリスト1-3で見てみましょう。

リスト1-3 変数の再代入

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int age = 20;
4     System.out.println("私の年齢は" + age);
5     age = 31;
6     System.out.println("…いや、本当の年齢は" + age);
7   }
8 }
```

変数ageを20で初期化（3行目）

変数ageに再度代入している（5行目）

実行結果

```
私の年齢は20
…いや、本当の年齢は31
```

この実行結果からわかるように、変数の内容は新たな値31で上書きされ、古い値20は消滅します。

変数の上書き

すでに値が入っている変数に代入をすると、古い値は消滅し、新しい値に内容が書き換わる。

しかし、プログラムを開発していると、「絶対に上書きされたくない」「内容が書き換えられたら困る」場合もあります。次のコードを見てください。

リスト1-4　書き換えてはいけない変数の値を上書きしてしまった例

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      double tax = 1.08;        ← 消費税を入れた変数
4      int fax = 5;              ← faxは5万円
5      System.out.println("5万円から4万円に値下げします");
6      tax = 4;                  ← 誤り！代入すべきはfax変数
7      System.out.println("ＦＡＸの新価格（税込み）");
8      System.out.println(fax * tax + "万円");
9    }
10 }
```

実行結果

```
5万円から4万円に値下げします
ＦＡＸの新価格（税込み）
20.0万円
```

「値下げします」って表示しながら、大幅値上げしちゃってますね。

でも、これが本物のネットショップのプログラムだったら、笑いごとじゃ済まないわよ。私もミスをしちゃいそう…。

6 行目で、fax 変数に代入すべきところを tax 変数に代入して上書きしてしまったため、計算結果がおかしくなってしまいました。そもそも税率である tax は、**その内容が動作中に書き換わる必要のない変数**です。このような場合、変数 tax の宣言の前に **final** という記述を加えることで書き換えを防止できます。

final 付きで宣言された変数は**定数**（constant variable）と呼ばれ、宣言と同時に初期値が代入された後は、値を書き換えることはできません。

定数の宣言方法

final 型 定数名 = 初期値 ;

※定数名にはすべて大文字（「TAX」など）を用いることが一般的

先ほどのリスト 1-4 を修正すると次のようになります。

リスト 1-5　定数の例

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
```

```
3        final double TAX = 1.08;          定数として税率を設定
4        int fax = 5;
5        System.out.println("5万円から4万円に値下げします");
6        TAX = 4;          コンパイルエラーとなり誤りに気づく
7        System.out.println("ＦＡＸの新価格（税込み）");
8        System.out.println(fax * TAX + "万円");
9    }
10 }
```

これで変数宣言の文はレッスン終了だよ。

1.4 第1章のまとめ

この章では、次のようなことを学びました。

開発と実行の流れ

- Javaの文法に従いソースコードを作成する。
- ソースコードをコンパイラでコンパイルして、バイトコードに変換する。
- インタプリタはバイトコードをマシン語に変換しながらCPUを動かす。

開発の流れと基本構造

- ソースコードはブロックによる二重構造を持っている。
- 外側部分は形式的記述であり、内側に文を並べる。
- 読みやすいソースにするためコメントとインデントを活用する。

変数宣言の文

- 変数は「型 変数名;」で宣言して利用する。
- 変数名は基本的に自由だが、一定の制約がある。
- 変数には代表的な9つの型があり、用途に合わせて使い分ける。
- finalを付けて宣言された定数の値は書き換えられない。

1.5 練習問題

練習 1-1

次の文章の［　　　］に入る言葉を答えてください。

Java でプログラムを開発するためには、［　ア　］と［　イ　］というソフトウェアが必要です。［　ア　］は、私たちが Java の文法に沿って記述した［　ウ　］を［　エ　］に変換してくれます。［　イ　］は内部に持っている［　オ　］のしくみを使ってこれを解釈し、マシン語に変換して CPU が実行します。

練習 1-2

画面に次のような結果を表示するソースコードを作成してください。このとき、ソースコード内で 3 を変数 a に、5 を変数 b に入れ、その掛け算の結果を変数 c に入れて、最後に変数 c を表示してください。

縦幅 3 横幅 5 の長方形の面積は、15

練習 1-3

以下に示した 5 つの値を格納するために適した型を考え、「初期化」による宣言を行うソースコードを作成してください（画面に表示する必要はありません）。なお、変数名は自由に考えて構いませんが、Java のルールに従ったものにしてください。2 つ以上の型が考えられる場合は、そのどちらでも構いません。

① true　② '駆'　③ 3.14　④ 314159265853979L
⑤ " ミナトの攻撃！ 敵に 15 ポイントのダメージを与えた。"

1.6 練習問題の解答

練習 1-1 の解答

(ア)コンパイラ　(イ)インタプリタ　(ウ)ソースコード
(エ)バイトコード　(オ)JVM

練習 1-2 の解答

以下は解答例です（おおむね合っていれば正解で構いません）。

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int a = 3; int b = 5;
4     int c = a * b;
5     System.out.println("縦幅3横幅5の長方形の面積は、" + c);
6   }
7 }
```

2行に分けても構いません

練習 1-3 の解答

以下は解答例です（おおむね合っていれば正解で構いません）。

```
boolean result = true;
char favoriteCharacter = '駆';
double pi = 3.14;
long number = 314159265853979L;
String msg = "ミナトの攻撃！　敵に15ポイントのダメージを与えた。";
```

float pi = 3.14F でも可

Java の進化の歴史

Java が公開された 1996 年頃は C 言語などが主流で、Java はマイナーな言語の 1 つにすぎませんでした。その後、バージョンアップのたびに改良され、現在のように広く使われるようになりました。

① 3 度の大幅改良

Javaの歴史の中では3度の「大幅改良」がありました。1度目はバージョン 1.1 から 1.2 への改良です。1.2 は現在普及している Java の基礎を築いたバージョンです。2 度目はバージョン 1.4 から 1.5 への改良で、より利用しやすく敷居の低い言語にするために、文法の拡張や API の追加が行われました。そして 2014 年、ラムダ式等の機能を取り入れた 8.0 がリリースされました。

②変化したバージョン番号の振り方

1.4 から 1.5 へのバージョンアップが比較的大がかりであったため、公式のバージョン番号としては1.5ではなく5を使うことになりました。つまり、**バージョン 5 とバージョン 1.5 は同じものを指しています**。それ以降、6、7、そして 8 とバージョン番号は振られていますが、Java の内部では 1.7 や 1.8 という古いバージョン表記が利用されている部分もあります。

第2章

式と演算子

第1章では「プログラムの中には文を並べて書く」こと、
また「変数の宣言」というJavaにおける2つのルールを学びました。
本章では、さまざまな計算を行うための「式と演算子」と、
キーボードから文字を入力したり、画面に文字を出力したり、
さらに乱数を生み出すなどの「命令実行の文」を学んでいきます。

CONTENTS

2.1 計算の文

2.1.1 計算の文とは?

第 1 章で紹介があった 3 種類の文のうち、「①変数宣言の文」はもうバッチリです♪

2 つ目の文は「②計算の文」でしたよね？

そうだよ。この文は「電子計算機」であるコンピュータにとって、とても大事な文なんだ。

計算の文とは、変数や値を用いたさまざまな計算処理をコンピュータに行わせるための文です。計算処理と言っても、いわゆる四則演算だけではありません。変数に値を代入することもコンピュータにとっては計算の一種なのです。

リスト 2-1　変数宣言の文と計算の文

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int a;
4      int b;
5      a = 20;
6      b = a + 5;
7      System.out.println(a);
8      System.out.println(b);
9    }
10 }
```

- 3行目：①変数宣言の文
- 5行目：②計算の文（代入）
- 6行目：②計算の文（足し算して代入）

実行結果

```
20
25
```

6 行目の「b = a + 5」のようなものを**式**(expression)と呼びます。見た目は数学の式のようですね。

数学… あぁ、その言葉を聞くだけで寒気がします…。

Java の式は数学より、ずっと簡単だから大丈夫だよ。

2.1.2 式の構成要素

式が何からできているか、分解して見てみよう。

式「b = a + 5」を分解すると、変数「a」「b」や値の「5」、そして「+」「=」の計算記号に分けることができます。Java を含む、多くのプログラミング言語では、この「a」「b」「5」を**オペランド**(operand)、そして「+」「=」を**演算子**(operator)と呼びます。これは簡単な式の例ですが、より複雑な式であっても同じで、**すべての式はこの 2 つの要素だけで構成されています**。

「+」や「=」といった演算子は式に含まれるオペランドを使って計算を行います。たとえば+演算子は「自分の左右にあるオペランドを加算する」機能を持っています(図 2-1)。Java に、どのような演算子があり、それがどのような機能を持つかについて、詳しくは 2.4 節で解説するとして、まずは次節でオペランドについての理解を深めていきましょう。

**図 2-1 式は演算子と
オペランドで構成されている**

2.2 オペランド

2.2.1 リテラル

オペランドって「変数や値」と考えておけばいいですか？

だいたい合っているよ。でも、より明確にするため
特に重要なオペランドである「リテラル」を紹介しよう。

オペランドの中でも数字「5」や文字列「Hello World」など、ソースコードに記述されている値のことを**リテラル**（literal）と呼びます。そして、それぞれの**リテラルはデータ型を持っています**（表 2-1）。そのリテラルが、どの型の情報を表すかはリテラルの表記方法で決まります。

表 2-1　代表的なリテラルの表記方法とデータ型

リテラルの種類	表記例	型
小数点がない数字（整数）	30	int
小数点がない数字で末尾が L または l（大きな整数）	300000L	long
小数点付きの数字（精度の高い小数）	30.5	double
小数点付きの数字で末尾が F または f（比較的精度の低い小数）	30.5F	float
true（真）または false（偽）	true	boolean
引用符で囲まれた文字	'雅'	char
二重引用符で囲まれた文字列	"Java"	String

一見、「'A'」と「"A"」は同じもののように見えるが、引用符の違いにより別のデータ型として扱われるので注意しよう。前者は char 型の**文字**「A」で、後者は String 型の**文字列**「A」だ。

「1」「'1'」「"1"」は、どれも別物ですね。気をつけなきゃ。

整数リテラルに関する応用記法

整数リテラルの先頭に 0x を付けると 16 進数、0 を付けると 8 進数、0b を付けると 2 進数として解釈されます。たとえば「int a = 0x11; int b = 0b0011;」と書くと、変数 a には 17、変数 b には 3 が代入されます。

また、リテラル中の任意の場所にアンダースコア記号（_）を含めることが許されています。日常生活で「2,000,000 円」などとカンマを入れるように、「long price = 2_000_000;」のように表記することで、大きな数値もわかりやすく表記することができます。

2.2.2 エスケープシーケンス

String 型や char 型のリテラルを記述する際に、ときどき用いられるものが**エスケープシーケンス**（escape sequence）と呼ばれる特殊な文字です。これは次のような「¥ 記号と、それに続く 1 文字」の合計 2 文字による記述方法で、その 2 文字で特殊な 1 文字を表現します。

表 2-2 代表的なエスケープシーケンス

表記	意味
¥"	二重引用符記号（"）
¥'	引用符記号（'）
¥¥	円記号（¥）
¥n	改行（制御文字）

な、なにこれ？ どうしてこんなものが必要なんですか？

「引用符記号」や「金額」を画面に表示するときなどに必要になる記号だよ。

たとえば「私の好きな記号は二重引用符（"）です」という文字列を画面に表示するプログラムを考えてみましょう。単純に考えて次のリスト 2-2 のように表記してしまうと、コンパイルエラーになってしまいます。

リスト 2-2　エスケープシーケンスを用いていない例（エラー）

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     System.out.println("私の好きな記号は二重引用符（"）です");
4   }
5 }
```

この部分だけが文字列と見なされる

Java は **2 つの二重引用符に囲まれた部分**を文字列リテラルと見なします。そのためリスト 2-2 の 3 行目では、途中までが文字列と解釈され、最後の「）です "」は文字列とは見なされません。よって、この部分がコンパイルエラーとなります。

Java は気がきかないなぁ。途中の「"」は文字列の終わりを表す記号じゃなくて、画面に出す文字としての「"」なのに…。

このような場合、エスケープシーケンスを用いれば、「画面に出す文字としての " である」ことを Java に対して伝えることができます（図 2-2）。

先ほどのプログラムは、リスト 2-3 のように改良することで期待どおりに動作します。

String msg = "私の好きな記号は"です。"

この部分のみが文字列と見なされてしまう

¥"で「"」一文字の代わりになる

String msg = "私の好きな記号は¥"です。"

エスケープ記号(¥)により途中の二重引用符も文字列として見なされる

図 2-2　文字列中に二重引用符を含めるにはエスケープシーケンスを用いる

リスト 2-3　エスケープシーケンスを用いた例

Main.java

```
1  public class Main {
2    public static void main(String[ ] args) {
3      System.out.println("私の好きな記号は二重引用符（¥"）です");
4    }
5  }
```

¥" によって二重引用符は文字と見なされる

このコードをコンパイルして実行すると、画面には次のように表示されます。

実行結果

```
私の好きな記号は二重引用符（"）です
```

金額を表示するとき、たとえば「¥1200」と表示したい場合は、「¥¥1200」というリテラル表記をすることで正しく¥マークが表示されるんだ。

2.2.3 リテラル以外のオペランド

式のオペランドとして利用できるのはリテラルのほかにも「変数」「定数」「命令の実行結果」などがあります。変数や定数については、すでに第1章で学習しましたね。また「命令の実行結果」については、2.6節で詳しく紹介します。

オペランドの学習は、ひとまずここで切り上げ、次節では「どのような順番で式が計算されていくか」を学びましょう。

評価のしくみ

2.3.1 評価の結果

「a = 3 + (b + c) * 4 ;」のような複雑な式も、Java はちゃんと実行してくれるんですね。

そうだよ。
Java が、どのような手順で式を計算しているかを説明しよう。

Java が式に従って計算処理をすることを、式の**評価**（evaluation）と呼びます。Java は 3 つの単純な原則に従いながら、式の一部から少しずつ部分的に処理してゆき、最後には式全体の計算処理が完了します。

評価の 3 つの原則って何ですか？

まず最も重要な「評価結果への置換の原則」を紹介しよう。
これは必ず理解してほしい。

評価結果への置換の原則

演算子は周囲のオペランドの情報を使って計算を行い、それら**オペランドを巻き込んで結果に化ける（置き換わる）**。

たとえば「1 + 5」という式の場合、＋演算子はオペランド1と5を使い、それらを足した計算結果「6」に化けます。

図2-3　演算子はオペランドを巻き込んで結果に「化ける」

より複雑な式「1 + 5 - 3」の場合には、段階的に評価が行われていきます。まず「1 + 5」の部分が「6」に化けて「6 - 3」という式に変形されます。それが処理された結果の「3」に化けて計算は終了です。

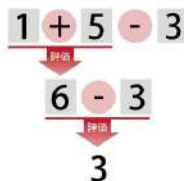



図2-4　「1 + 5 - 3」→「6 - 3」→「3」と変化する

2.3.2　優先順位

式に複数の演算子が含まれることもあると思いますが、どの部分から順に評価していくんですか？ 左の演算子から順番にかな？

いや、そうとは限らないよ。
演算子には「優先順位の原則」があるんだ。

優先順位の原則

式に演算子が複数ある場合は、**Javaで定められた優先順位の高い演算子から順に**評価される。

Java には多くの演算子がありますが、それらには**優先順位**(15 段階)が定められています(優先順位は次節で解説します)。たとえば「5 番目に優先」の演算子グループに属している＋演算子よりも、「4 番目に優先」のグループに属する＊演算子のほうが先に評価されます。「1 + 5 * 3」という式があれば、先に掛け算が行われるのです。もし、式の中で「1 + 5」の評価を優先したい場合は、丸カッコ「()」を使うことで評価順位を引き上げることができます。

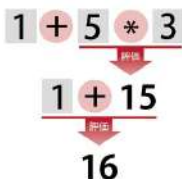
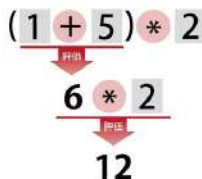



図 2-5　優先度が高い演算子から評価される

すべての演算子の優先順位をがんばって覚えなきゃ！

いや、そんな必要はないよ。プログラムを書いていれば自然に覚えられるから、必要なときに調べれば十分だよ。

2.3.3 結合規則

もし「同じ優先順位の演算子」が 2 つ以上あったら、どっちが優先されるんですか？

そのルールは「結合規則の原則」で決められているんだ。

結合規則の原則

式の中に同じ優先順位グループに属する演算子が複数ある場合、**演算子ごとに決められた「方向」から順に**評価される。

すべての演算子には、左から評価をするか、または右から評価をするかという「方向」が**結合規則**として定められています。たとえば＋演算子は左から右へ評価しますので「10 + 5 + 2」は次のようになります。

①「10 + 5」を評価して結果は「15」。
②「15 + 2」を評価して結果は「17」。

一方、＝演算子は右から左へ評価しますので「a = b = 10」という式の場合は次のようになります。

①最も右の = 演算子に関する「b = 10」が評価され、b に 10 が代入された結果、式自体は「10」に化ける。
②次に「a = 10」が評価され、a に 10 が代入される。

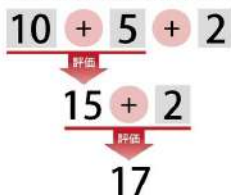
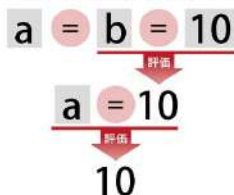



図 2-6　結合規則に従って評価される

それぞれの演算子に定められている「優先順位」と「結合規則」については次節で紹介しよう。

2.4 演算子

＋や＊といった演算子が出てきましたが、他にはどんな演算子があるんですか？

Java には多くの演算子が定められているよ。その中から代表的なものを紹介していこう。

2.4.1 算術演算子

左右の数値オペランドを使って四則計算を行うための演算子は、**算術演算子**と総称されています。以下の 5 つを覚えておきましょう。

表 2-3　代表的な算術演算子

演算子	機能	優先順位	評価の方向	評価の例
+	加算（足し算）	中（5）	左→右	3 + 5 → 8
-	減算（引き算）	中（5）	左→右	10 − 3 → 7
*	乗算（掛け算）	高（4）	左→右	3 * 2 → 6
/	除算（割り算） （※整数演算では商）	高（4）	左→右	3.2 / 2 → 1.6 9 / 2 → 4
%	剰余（割り算の余り）	高（4）	左→右	9 % 2 → 1

注意が必要なのは除算演算子（/）です。この演算子は割り算を行うものですが、**整数同士の割り算に用いると「商」**を計算します。「9 / 2」が 4 と評価されてしまうのが困る場合は、「9.0 / 2」のようにどちらかのオペランドを小数にします。

除算演算子の落とし穴には十分、注意してほしい。

2.4.2 文字列結合演算子

左右の文字列オペランドを結合して1つの文字列にする演算子です。加算演算子と同じ+記号を使います。

表 2-4 文字列結合演算子

演算子	機能	優先順位	評価の方向	評価の例
+	文字列の連結	中(5)	左→右	"こん"+"にちは"→"こんにちは" "ベスト"+3→"ベスト3"

これも第1章から使ってきた演算子ですよね?

そうだね。ただし、文字列以外との結合については少し注意点があるから、2.5.4項で詳しく説明するよ。

2.4.3 代入演算子

右オペランドの内容を左オペランドの変数に代入する演算子です。演算や結合を行いながら代入を行うものもあります。いずれも優先順位が最低なので「代入は基本的に最後に行われる」と覚えておけばよいでしょう。

表 2-5 代表的な代入演算子

演算子	機能	優先順位	評価の方向	評価の例
=	右辺を左辺に代入	最低(15)	右→左	a=10→a(中身は10)
+=	左辺と右辺を加算して左辺に代入	最低(15)	右→左	a+=2→a (a=a+2と同じ)
-=	左辺から右辺を減算し左辺に代入	最低(15)	右→左	a-=2→a (a=a-2と同じ)
*=	左辺と右辺を乗算し左辺に代入	最低(15)	右→左	a*=2→a (a=a*2と同じ)
/=	左辺と右辺を除算し左辺に代入	最低(15)	右→左	a/=2→a (a=a/2と同じ)
%=	左辺と右辺を除算し、その余りを左辺に代入	最低(15)	右→左	a%=2→a (a=a%2と同じ)
+=	左辺の後に右辺を連結して代入	最低(15)	右→左	a+="風"→a (a=a+"風"と同じ)

たとえば変数aの内容を3増やしたい場合は「a += 3」と「a = a + 3」の2種類の書き方があり、どちらを用いても構いません。

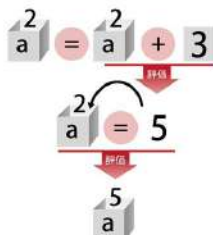



図2-7　「a = a + 3」の評価の流れ

「a = a + 3」という数式に少し違和感があるけど、評価の流れを考えれば納得できるね。

2.4.4 インクリメント・デクリメント演算子

前節で紹介したように、変数aの内容を3増やしたい場合には「a = a + 3」、あるいは「a += 3」と書くことができます。しかし、3ではなく1だけ増やしたり減らしたりする場合には、さらに便利な記述方法が準備されています。

表2-6　インクリメント・デクリメント演算子

演算子	機能	優先順位	評価の方向	評価の例
++	値を1増やす	最高（1）	左→右	a++ → a （a = a + 1 や a += 1 と同じ）
--	値を1減らす	最高（1）	左→右	a-- → a （a = a - 1 や a -= 1 と同じ）

リスト2-4　インクリメントとデクリメント

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int a;
4      a = 100;
```

```
5        a++;        ← aの内容が1増える
6        System.out.println(a);
7    }
8  }
```


実行結果

```
101
```

この演算子は左右両方にはオペランドを持たないんですね。

そうだね。1つしかオペランドを持たない演算子は、ほかにもあって「**単項演算子**」と総称されているよ。

++や--は、ほかの演算子と一緒に使わない!

インクリメント・デクリメント演算子は「++a」のようにオペランドの前に付けることもできます。前または後、どちらの表記法を利用しても変数aの中身が1増えることには違いありません。しかし、ほかの演算子と一緒に利用すると++aとa++では微妙な違いが生じます。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int a = 10;
4      int b = 10;
5      System.out.println(++a + 50);
6      System.out.println(b++ + 50);
7    }
8  }
```

実行結果

```
61
60
```

変数 a も変数 b も初期値は 10 です。これに 1 を足し、50 を加えた値を表示するよう指示していますが、結果が異なっている点に注意してください。この動作の違いを理解するために、5 行目と 6 行目が次のように実行されるかを見てみましょう。

● 5 行目の実行のされ方
①変数 a の値が 1 増える。
②それに 50 を加えたものが画面に表示される。

● 6 行目の実行のされ方
①変数 b に 50 を加えたものが画面に表示される。
②変数 b の値が 1 増える。

このように他の演算と組み合わせた場合、インクリメント・デクリメント演算子がオペランドの前にあるか後にあるかで「1 増える（1 減らす）タイミング」が変わってきます。そのため、これをほかの演算子と一緒に使うと不要なバグの原因になります。特別な理由がない限り、リスト 2-4 のように単独で使うように心がけましょう。

型の変換

2.5.1 3種類の型変換

演算子の型って「いいかげん」なのかなって思うんですけど…。

確かに…。整数をdouble型変数に代入できちゃうし、「文字列と数字」を+で結合できちゃうし…。

これは式評価の過程で「型変換」というしくみが働いてくれているおかげなんだ。

前節で学んだ演算子の多くは、原則として左右のオペランドが同じ型であることを要求します。しかし、実際には「違う型に代入」や「違う型同士で計算」をさせても文法エラーにならないことがあります。そのためJavaは型について「いいかげん」に解釈して動いてくれているように感じることもあるでしょう。

```
double d = 3;
String s = "ベスト" + 3;
```

double型変数にint型の3を代入できてしまう

String型とint型を連結できてしまう

このような記述がエラーにならないのは、**Javaが式を評価する過程で自動的に型を変換している**からです。Javaには型を変換するしくみが3つ備わっていて、特に次の①と③はプログラマが気にしなくても自動的に機能します。

①代入時の自動型変換
②明示的な型変換
③演算時の自動型変換

次項から型変換のしくみについて1つずつ紹介していきます。

2.5.2 代入時の自動型変換

第 1 章の変数の解説で触れたように、**ある型で宣言された変数には、その型の値しか代入できません**（1.3.1 項）。int 型変数には int 型の整数だけ、String 型の変数には String 型の文字列だけしか代入できない、これが原則です。

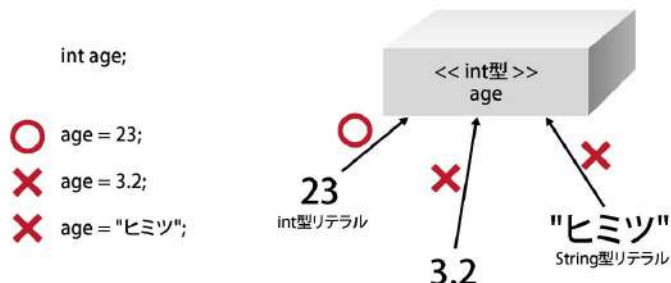



図 2-8　変数の型と値の型が一致しないと代入できない

たとえば long 型の値は int 型変数には代入できない。
数字が大きすぎて、箱に入りきれないかもしれないしね。

でも先輩、逆に int 型の値を long 型変数に入れるのは、int よりも long の箱のほうが大きいから実害ないんじゃないでしょうか？

いいことに気づいたね。そのとおりだよ。

Java の数値型は次の図 2-9 のように意味的な大小関係が定められています。そして、「小さな型」の値を「大きな型」の変数に代入する場合に限って、**値が自動的に箱の型に変換されて代入**されます。

このしくみがあるため、リスト 2-5 のような代入はコンパイルエラーになりません。

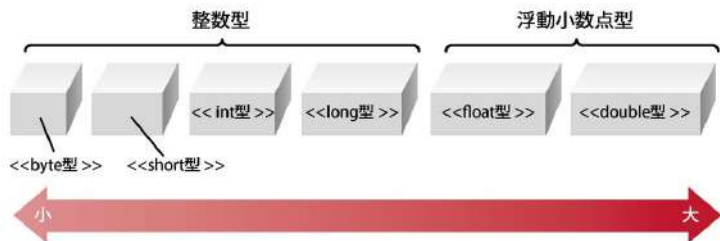



図 2-9　各数値型の意味的な大小関係

リスト 2-5　値より大きな型の変数に代入

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      float f = 3;
4      double d = f;
5      System.out.println(f);
6      System.out.println(d);
7    }
8  }
```

- 3行目：float 型の変数に int 型を代入
- 4行目：double 型に変数に float 型を代入

実行結果

```
3.0
3.0
```

リスト 2-5 の 3 行目では、リテラルの 3（int 型）は 3.0F（float 型）に自動的に変換されて変数 f に代入されています。同様に 4 行目も、float 型の変数 f が double 型に変換されてから変数 d に代入されます。

代入時の自動型変換

意味的に「小さな型」の値を「大きな型」の箱に代入する場合、代入される値が代入先の変数の型に自動的に変換されてから代入が行われる。

逆に「大きな型」の値を「小さな型」の変数に代入することは原則としてできません。箱に入りきらない可能性があるからです。リスト 2-6 をコンパイルすると、「精度が落ちている可能性」という文法エラーが表示されてコンパイルは失敗します。

リスト 2-6　データより小さな型の変数に代入（エラー）

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int i = 3.2;
4    }
5  }
```

小数点以下はどうなっちゃうの？

ただし、byte 型や short 型の変数に数値リテラル値を代入できないと困るため、「byte b = 3;」のように **int 型リテラルを byte 型や short 型の変数に対して実害がない範囲で単純代入すること**だけは例外的に認められています。この例外を含め、ここまで学んだ代入が可能かどうかを一覧表にまとめたものが図 2-10 です。

図 2-10　数値型に関する代入の可否

代入する値の型＼代入先の変数の型	byte	short	int	long	float	double
byte	◎	○	○	○	○	○
short	×	◎	○	○	○	○
int	△	△	◎	○	○	○
long	×	×	×	◎	○	○
float	×	×	×	×	◎	○
double	×	×	×	×	×	◎

◎……そのまま代入可能　○……自動型変換により代入可能　△……例外的・限定的に可能　×……代入できない

整数型としての char 型

char 型は文字を扱う型ですが、内部的には 0 〜 65535 の範囲の数値として情報を管理しています。厳密には int や short などのような整数型の一種であるため算術演算も行えますし、型変換も行われます。しかし、一部の用途を除いて char 型を数値として利用することはほとんどないため、本書では「数値型としての char 型の取り扱い方」についての説明は割愛しました。

2.5.3 強制的な型変換

すでに説明したように「大きな型」の値を「小さな型」の変数に代入することは原則としてできません。しかし、それを強制的に行う方法があります。プログラマが明示的に、「小さな型に変換して押し込め！」と指示をすれば Java は変換と代入を強行します。

リスト 2-7　強制的な型変換

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int age = (int) 3.2;
4     System.out.println(age);
5   }
6 }
```

3.2 を int に型変換して代入せよ！

実行結果

```
3
```

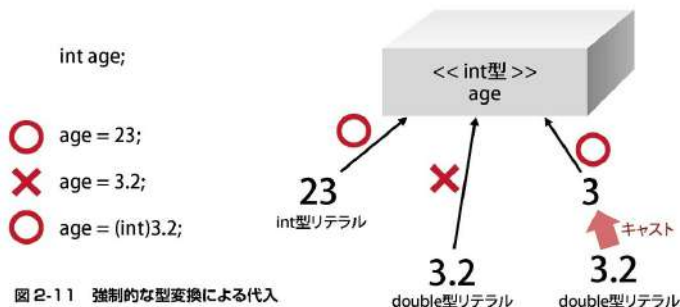
3.2 という double 型リテラルの前に記述された「(int)」が強制的な型変換を指示する**キャスト演算子**（cast operator）です。

キャストによる強制的な型変換

(変換先の型名)式

キャスト演算子は、**元のデータの一部を失っても**データを強制的に変換しようとします。「子ども用の小さなお弁当箱に、36cmの大判ピザを無理矢理詰め込む」ようなものですから、入りきれない部分(情報)ははみ出てしまいます。

はみ出た部分は捨てられてしまい、情報の欠損が発生します。先ほどのリスト2-7では、「3.2」を強引にint型へ変換したために小数点以下の情報が失われてしまいました。

図 2-11　強制的な型変換による代入

キャストは乱暴な道具なので利用に代償を伴います。キャストを用いても変換できない型の組み合わせも存在しますし、データの欠損が不具合につながることもあります。最終手段として、どうしても必要な場合もありますが、**よほどの理由がない限り使わない**と覚えておいてください。

2.5.4 演算時の自動型変換

それでは、最後に3つ目の型変換のしくみを説明しよう。

代入だけではなく算術演算子などによって計算が行われる場合も「左右のオペランドは同一の型」が原則です。たとえば除算演算子 (/) による割り算のようすを見てみましょう。

図 2-12　同じ型同士で演算を行った場合

算術演算の結果は、計算で使用されたオペランドの型となります。つまり、int 型同士で計算した場合は int 型の結果、double 型同士で計算した場合は double 型の結果になります。

では、異なる型で演算を行った場合はどうなるでしょうか？ その場合には**「意味的に大きな型」に統一されてから演算**が行われます。

図 2-13　異なる型同士で演算をした場合

数値型同士の演算時型変換のルール

- 片方のオペランドが double なら、他方を double に型変換して揃える。
- 片方のオペランドが float なら、他方を float に型変換して揃える。
- 片方のオペランドが long なら、他方を long に型変換して揃える。
- 片方のオペランドが int なら、他方を int に型変換して揃える。
- short や byte のオペランドは int に型変換される。

代入にしても演算にしても、Java はきっちりと型を揃えてから処理するんですね。

そうだよ。実際に演算時に型変換を行うサンプルも見ておこうか。

リスト 2-8　異なる型同士の算術演算

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     double d = 8.5 / 2;
4     long l = 5 + 2L;
5     System.out.println(d);
6     System.out.println(l);
7   }
8 }
```

2（int 型）を 2.0（double 型）に変換

5（int 型）を 5L（long 型）に変換

実行結果

```
4.25
7
```

「int と long」のように数値型同士の演算でなく、「int と String」のように「数値型と文字型」の組み合わせでも、別のルールで自動的に型が変換される。これも紹介しておこう。

オペランドが数値型と String 型の場合は、次のような型変換が自動的に行われます。

文字列を含む演算時型変換

片方のオペランドが String なら、他方も String に変換して連結する。

リスト 2-9 文字列の連結

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String msg = "私の年齢は" + 23;
       23 (int 型) が "23" (String 型) に変換されて連結される
4      System.out.println(msg);
5    }
6  }
```

実行結果

```
私の年齢は23
```

おめでとう、これで「②計算の文」はすべてマスターしたよ。最後の「③命令実行の文」に進もう。

Java言語仕様をのぞいてみよう

本書では初めてJavaを学ぶ人にもわかりやすいように文を3種類に分類して解説しています。しかし、Javaにおける文の厳密な分類はとても複雑です。Javaの正式な決まりは「Java言語仕様（The Java Language Specification）」にまとめられています。書籍やWebサイトで閲覧できますので、ぜひ参照してください。

2.6.1 命令実行の文とは

「②計算の文」は演算子とか型変換とか、たくさん出てきて結構、複雑でした…。

お疲れさま。でも喜んでほしい。最後の１つ「③命令実行の文」はとてもカンタンで、しかも楽しいよ。

ここで図1-14（p.43）を、もう一度見てください。ここまでJavaにおける3種類の文のうち2種類を解説してきました。最後に残っているのは「③命令実行の文」です。

命令実行の文は**Javaが準備してくれているさまざまな命令を呼び出す**ための文です。この文を使えば、「足し算」や「代入」より、ずっと高度な処理をコンピュータに行わせることができます。最も代表的なものとして、おなじみの「System.out.println」があります。

リスト 2-10　命令（画面出力）実行の文

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String name = "すがわら";                 ①変数宣言の文
4      String message;
5      message = name + "さん、こんにちは";      ②計算の文
6      System.out.println(message);             ③命令実行の文
7    }
8  }
```

実行結果

すがわらさん、こんにちは

ちなみに、リスト 2-10 は 5 行目と 6 行目を 1 つの文にまとめて、「System.out.println(name + " さん、こんにちは ");」にもできる。

命令実行の文の中で式を使うこともできるんですね。

命令実行の文は、末尾に丸カッコで囲まれた部分が登場するのが特徴です。

命令実行の文

呼び出す命令の名前 (引数);

カッコの中に記述するものは**引数**や**パラメータ**と呼ばれるもので、その命令を呼び出すにあたって必要となる追加情報です。System.out.println() であれば、「何を画面に表示するか」という情報を引数で指定します。

Java で利用できる命令は System.out.println() 以外にも数多くありますが、引数を 2 つ指定するものや、1 つも指定しなくてもよいものなど、それぞれ引数の種類や数が異なります。

ほかにはどんな命令があるんですか？
一発でゲームが作れちゃうような命令とかあるんですか？！

まぁまぁ落ち着いて。少しずつ紹介していくから、
楽しみにしていてほしい。

使える命令が System.out.println() だけでは楽しくありませんね。Java には「音

を鳴らす」「ファイルに書き込む」「キーボードから入力を受け付ける」「プリンタに文字を印刷する」「ネットワークで通信を行う」など、数多くの命令が準備されています。しかし、現時点の私たちには、それらすべてを使いこなすことは難しいので、次項以降では使いやすい命令を少しずつ紹介していきます。

なお、紹介した命令について書き方を丸暗記する必要はありません。後から使いたくなったときに「そういえばこういう命令があったはず」と思い出して本書を読み返せば大丈夫です。気楽に読み進めてください。

2.6.2 画面に文字を表示する命令

まずは基礎中の基礎、画面に文字を表示する命令からいこう。
System.out.println() と似た命令をもう 1 つ紹介しよう。

改行せずに画面に文字を表示する

```
System.out.print( ① );
```

※①は画面に表示したい値や式

System.out.println() 命令とよく似ている命令に System.out.print() 命令があります。この命令は画面に①の内容を表示しますが、表示後に改行しません。このため、連続して呼び出すと表示内容が連続して表示されます。

リスト 2-11　改行なし画面出力の命令

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String name = "すがわら";
4      System.out.print("私の名前は");
5      System.out.print(name);
```

```
6       System.out.print("です");
7     }
8   }
```


実行結果

私の名前はすがわらです

2.6.3 大きいほうの数字を代入する命令

2つの値を比較して大きいほうの数字を代入する

int m = Math.max (①, ②);

※①および②は比較したい値や式

Math.max() 命令は、2つの引数を指定して呼び出す命令です。引数として与えた①と②のうち、大きなほうの値が m に代入されます。なお、解説の都合で変数名として m を使っていますが、ほかの変数名でも構いません。

リスト 2-12　大きいほうの数字を選択させる命令

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int a = 5;
4     int b = 3;
5     int m = Math.max(a, b);
6     System.out.println("比較実験：" +
          a + "と" + b + "とで大きいほうは…" + m );
7   }
8 }
```

実行結果

```
比較実験：5と3とで大きいほうは…5
```

「Math.max (5,3) 」の結果が 5 になっているんですね。

「Math.max (5,3)」が 5 に化ける…。これ、「評価」ですか？

鋭いね。実は「命令の実行」も式の一種なんだ。

2.6.4 文字列を数字に変換する命令

文字列を数字に変換する

```
int n = Integer.parseInt ( ① );
```

※①は数字として解釈させたい文字列 ("23" など)

たとえば String 型変数に入っている "10" は文字列なので、そのままでは四則演算ができません。**文字列の "10" を数字の 10 に変換して計算を行いたい**場合には、この命令を使ってください。

命令の①に「整数として読むことができる」文字列が入った String 型の変数やリテラルを指定すると、int 型の整数に変換して n に代入してくれます。

リスト 2-13　String 型を int 型に変換する命令

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String age = "31";
```

```
4       int n = Integer.parseInt(age);
5       System.out.println
            ("あなたは来年、" + ( n + 1 ) + "歳になりますね。");
6   }
7 }
```


実行結果

```
あなたは来年、32歳になりますね。
```

もし①に「こんにちは」のような「数字ではない文字列」を指定すると、プログラム実行中にエラーが起きて異常終了するから気をつけてほしい。

2.6.5 乱数を生み出して代入する命令

湊くんが大好きなゲームに不可欠なのが乱数だ。

ランスウ……？

コンピュータの中に入っているサイコロみたいなものよ。毎回ランダムに違う値が取り出せるの。

乱数を発生させる

```
int r = new java.util.Random().nextInt( ① );
```

※①は発生させる乱数の上限値（指定値自体を含まない）

①に 1 以上の整数を指定してこの命令を呼び出すと、0 以上かつ①で指定した数字未満のランダムな整数が r に代入されます。r に何が代入されるかは実行するまでわかりません。①に 10 を指定すると r には 0 〜 9 のいずれかが代入されます。

リスト 2-14　ランダムな数を生成する命令

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int r = new java.util.Random().nextInt(90);
4      System.out.println("あなたはたぶん、" + r + "歳ですね？");
5    }
6  }
```

実行結果

```
あなたはたぶん、31歳ですね？
```

2.6.6　キーボードから 1 行の入力を受け取る命令

あとゲーム作りに必要なのは「キーボードから文字を入力する」命令だね。

キーボードから 1 行の文字列の入力を受け付ける

```
String input = new java.util.Scanner ( System.in ).
nextLine ();
```

キーボードから 1 つの整数の入力を受け付ける

```
int input = new java.util.Scanner ( System.in ).
nextInt ();
```

これらの文を実行すると、プログラムは一時停止状態になり、利用者がキーボードから文字を入力できるようになります。そして、利用者が文字列をキーボード入力してEnterキーを押すと、その内容が変数inputに代入されます。nextLine()は文字列を、nextInt()は数字の入力を受け取るために使います。

リスト2-15　キーボードから入力を受け付ける命令

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("あなたの名前を入力してください。");
4      String name = new java.util.Scanner(System.in).nextLine();
5      System.out.println("あなたの年齢を入力してください。");
6      int age = new java.util.Scanner(System.in).nextInt();
7      System.out.println
           ("ようこそ、" + age + "歳の" + name + "さん");
8    }
9  }
```

実行結果

```
あなたの名前を入力してください。
すがわら⏎        キーボードから名前を入力
あなたの年齢を入力してください。
31⏎              キーボードから年齢を入力
ようこそ、31歳のすがわらさん。
```

もうキーボード入力も、乱数生成もできるようになりました。

上手に組み合わせたら、簡単な占いゲームとか作れますね。

これまで習った命令・式・演算子を使って、ぜひ自分なりにプログラムを書いてほしい。それが上達の近道だからね。

2.7 第 2 章のまとめ

この章では、次のようなことを学びました。

式

- 式は演算子とオペランドで構成されている。
- リテラルにも型があり記述方法で決定される。
- 演算子が評価されると、その演算子とオペランドは結果に化ける。
- 演算子は優先順位と結合規則に従い評価される。

型変換

- 大きい変数に小さなデータを代入する際、自動的に型が変換され代入される。
- 小さな変数に大きなデータを代入する際、キャストを行うことで代入できる。
- 式の評価時、大きなデータに揃えるよう自動的に型が変換される。

命令の実行

- Java に用意されている、さまざまな命令を実行することができる。

2.8 練習問題

練習 2-1

次のようなプログラムがあります。

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int x = 5;
4     int y = 10;
5     String ans = "x+yは" + x + y;
6     System.out.println(ans);
7   }
8 }
```

このプログラムを実行すると以下の結果が表示されます。

```
x+yは510
```

「x+y は 15」と表示させたいのですが、意図どおりに動きません。正しく動作するように修正してください。

練習 2-2

次の中で文法として正しいものを、すべて選んでください。

① int x = 3 + 5.0;
② double d = 2.0F;
③ int i = "5";
④ String s = 2 + " 人目 ";
⑤ byte b = 1;
⑥ double d = true;
⑦ short s = (byte)2;

練習 2-3

以下の内容のプログラムを作成してください。

①画面に「ようこそ占いの館へ」と表示します。
②画面に「あなたの名前を入力してください」と表示します。
③キーボードから 1 行の文字入力を受け付け、String 型の変数 name に格納します。
④画面に「あなたの年齢を入力してください」と表示します。
⑤キーボードから 1 行の文字入力を受け付け、String 型の変数 ageString に格納します。
⑥変数 ageString の内容を int 型に変換し、int 型の変数 age に代入します。
⑦ 0 から 3 までの乱数を生成し、int 型の変数 fortune に代入します。
⑧ fortune の数値をインクリメント演算子で 1 増やし、1 から 4 の乱数にします。
⑨画面に「占いの結果が出ました！」と表示します。
⑩画面に「（年齢）歳の（名前）さん、あなたの運気番号は（乱数）です」と表示します。その際に（年齢）には変数 age を、（名前）には変数 name を、そして（乱数）には⑧で作った数字を表示させます。
⑪画面に「1: 大吉 2: 中吉 3: 吉 4: 凶」と表示します。

2.9 練習問題の解答

練習 2-1 の解答

5 行目を次のように修正します。

```
String ans = "x+yは" + (x + y);
```

不具合の原因は 5 行目の最後の「x + y」を丸カッコ「()」で囲っていなかったからです。「2.5.4 演算時の自動型変換」で解説したように、オペランドの中に文字列が含まれると、そのほかのオペランドも文字列型に変換されます。そのため int 型の変数である x と y の内容は文字列型に変換され「文字列として連結」されます。その結果、画面に「x+y は 510」と表示されたのです。

意図どおりに「x+y は 15」と表示させるには、x と y を丸カッコで囲い、この計算の評価順位を引き上げる必要があります。

練習 2-2 の解答

正しい文は②、④、⑤、⑦です。

練習 2-3 の解答

Main.java

```
public class Main {
  public static void main(String[] args) {
    System.out.println("ようこそ占いの館へ");
    System.out.println("あなたの名前を入力してください");
    String name = new java.util.Scanner(System.in).nextLine();
    System.out.println("あなたの年齢を入力してください");
    String ageString =
        new java.util.Scanner(System.in).nextLine();
    int age = Integer.parseInt(ageString);
    int fortune = new java.util.Random().nextInt(4);
    fortune++;
    System.out.println("占いの結果が出ました！");
    System.out.println(age + "歳の" + name +
        "さん、あなたの運気番号は" + fortune + "です");
    System.out.println("1: 大吉  2:中吉  3:吉  4: 凶");
  }
}
```

第3章

条件分岐と繰り返し

私たちは日々、条件に応じた行動の分岐や繰り返しをしています。
たとえば「もしも天気予報が雨だったら傘を持っていく」という分岐や、
「正解するまで何度も問題を解く」という繰り返しです。
プログラムも、これと同じように条件分岐と繰り返しを行いながら
処理を進めていきます。この章では、条件分岐や繰り返しを実現するための
Javaの構文を学んでいきます。

CONTENTS

3.1 プログラムの流れ

3.1.1 代表的な制御構文

第1章と第2章では、変数や型・リテラル・演算子などを使用した文の書き方を学習しました。そして、それらの文は「上から順に1つずつ」実行されることがルールでした。文を実行させる順番のことを**制御構造**（または制御フロー）といい、代表的なものとして「**順次**」「**分岐**」「**繰り返し**」の3つがあります。

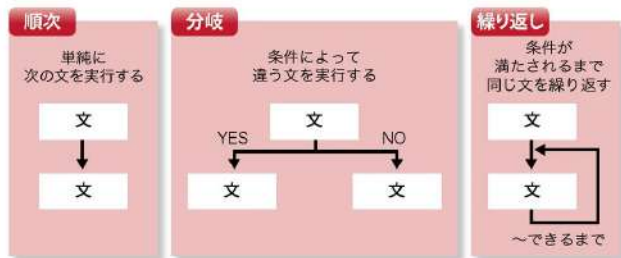



図 3-1　代表的な制御構造。「順次」と「分岐」、そして「繰り返し」（ループ）がある

第2章でも使ってきた「上から順に1つずつ」の流れは、ここでいう「順次」のことですね。

そうだよ。そして、この章では残りの2つを学ぶんだ。

「順次」、「分岐」、「繰り返し」の各制御構造を組み合わせることで、文の実行の流れを自由に作れるようになります。

構造化定理

どんなに複雑なプログラムでも、順次・分岐・繰り返しの 3 つの制御構造を組み合わせれば作成することが可能なことが数学的に証明されています。

プログラムを作るには「順次」「分岐」「繰り返し」を覚えれば何とかなる！ ということですね

そうだね。逆に言えば、この 3 つの制御構造をマスターしてうまく使えなければ本格的なプログラムは作れないんだ。

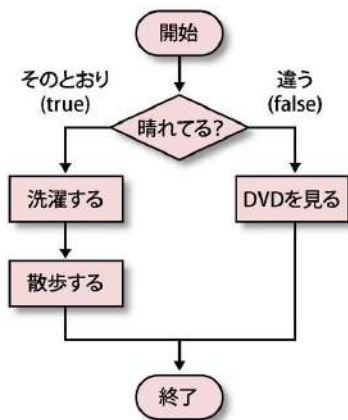
3.1.2 分岐を体験する

まずはシンプルな例を見て、雰囲気をつかんでおこう。

私たちは日常生活でも、条件によって行動を変化させています。次の文章を見てください。

もし、明日が晴れなら、洗濯してから散歩にいこう。
でも、明日が雨だったら、部屋で DVD を見ていよう。

これは、「洗濯する」「散歩する」「DVD を見る」など、それぞれの行動の流れを「晴れ」または「雨」という**条件によって変化させている**わけです。この行動をフローチャートで表現すると次のようになります。

図 3-2　天気による行動の変化をフローチャートで表したもの

この分岐を Java のコードで表現すると、次のようになります。

リスト 3-1　天気による行動の変化を Java で表したもの

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      boolean tenki = true;
4      if (tenki == true) {
5        System.out.println("洗濯をします");
6        System.out.println("散歩にいきます");
7      } else {
8        System.out.println("DVDを見ます");
9      }
10   }
11 }
```

もし tenki 変数の中身が true だったら…

そうでなければ…

リスト 3-1 の実行結果
(3 行目で true を代入したとき)

```
洗濯をします
散歩にいきます
```

リスト 3-1 の実行結果
(3 行目で false を代入したとき)

```
DVDを見ます
```


図 3-2 のフローチャートとリスト 3-1 のコードを見比べてください。コードの各部分が、どのような意味やしくみになっているか次のように想像できます。

- **if という命令を使えば「分岐」を行うことができる(「if」は「もしも」という意味の英単語)。**
- **if の後ろの () 内には「晴れているか？」などの「分岐条件」を書く。**
- **変数 tenki が true かどうかのチェックを行うには「==」(イコール記号 2 つ)を使う。**
- **分岐条件が成立していたら、() の直後にあるブロック("{" と "}" で囲まれた部分)の中身だけを実行する。**
- **分岐条件が成立していなければ、else の後ろにあるブロック("{" と "}" で囲まれた部分)の中身だけを実行する。**

このように「if」を使用した文のことを「if 文」と言うんだ。

if 文って英語の文章みたいですね。

本当ね。else は「そうでなければ」という意味の英単語だから「もし○○ならば A をする。そうでなければ B をする」という感じね。

3.1.3 繰り返しを体験する

分岐の次に繰り返しを見てみましょう。日常生活において、次のような繰り返しを行うことがありますね。

もしトイレに誰か入っていたら「扉をノックして1分待つ」を繰り返す。

この行動は次のフローチャートで表すことができます。

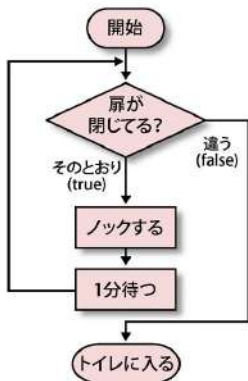



図3-3 日常生活の中の繰り返し処理

これをJavaのコードで表すと、以下のようになります。なお、3行目でtrueを代入して実行すると無限ループになりますので、Ctrl+Cで強制終了してください。

リスト3-2 トイレの空きを待つ繰り返し処理

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     boolean doorClose = true;    // ここでtrueかfalseを代入
4     while (doorClose == true) {
5       System.out.println("ノックする");
6       System.out.println("1分待つ");
7     }
8   }
9 }
```

ドアが閉まっている間は…

3行目がtrueの場合、
このプログラムを実行すると無限ループになりますので、
Ctrl + Cで強制終了してください。

リスト 3-2 の実行結果
(3 行目で true を代入したとき)

ノックする
1分待つ
ノックする
1分待つ
‥　　無限に繰り返しが続きます…。

リスト 3-2 の実行結果
(3 行目で false を代入したとき)

何も表示されない。

リスト 3-2 のコードと図 3-3 のフローチャートを見比べると次のことに気づくでしょう。

- **while という命令を使えば「繰り返し」制御を行うことができる(while は「〜の間は」という意味の英単語)。**
- **while の後ろの () 内には「ドアが閉まっている」などの「繰り返しを続ける条件」を書く。**
- **繰り返しを続ける条件が成立している限り、何度でも直後のブロック("{" と "}" で囲まれた部分)の中身が繰り返し実行される。**

ちなみに繰り返しのことを「ループ」とも言うよ。

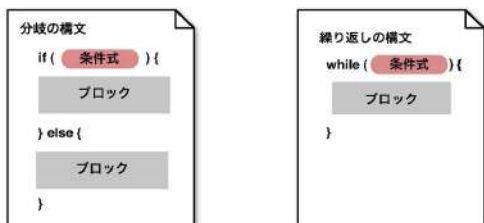
3.1.4 制御構文の構成要素

「分岐」の if 文と「繰り返し」の while 文は異なる制御構造の文なのに、書き方は似ていますね。

それは両方の構文が共に「条件式」と「ブロック」から構成されているからだよ。
ここまで見てきた if 文や while 文のような制御構造を表す文のことを、**制御構文**といいます。制御構文は次の 2 つの構成要素から成り立っています。

制御構文の構成要素

［条件式］　分岐条件や繰り返しを続ける条件を示した式
［ブロック］　分岐や繰り返しで実行する一連の文の集まり

図 3-4　制御構文は〔条件式〕と〔ブロック〕から成り立つ

つまり、「条件式」と「ブロック」の書き方を理解できれば制御構文を身に付けることができる。次節では、まずブロックの書き方を学ぼう。

3.2.1 ブロックとは

ブロックとは複数の文をひとまとまりとして扱うためのものです。ブロック中には複数の文を記述できます。ただし、次に紹介する「**ブロックにまつわる2つのルール**」を守る必要があるので、必ず覚えてください。

■ルール1:波カッコの省略

ブロックとは通常、波カッコ "{"}" で囲まれた部分を指していますが、**内容が1行しかなければ、波カッコを省略しても構わない**というルールがあります。たとえば、3.1.2項のリスト3-1は次のように記述しても同じ意味になります。

リスト3-3　波カッコを省略した記述

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      boolean tenki = true;    // ここでtrueかfalseを代入
4      if (tenki == true) {
5        System.out.println("洗濯をします");
6        System.out.println("散歩にいきます");
7      } else
8        System.out.println("DVDを見ます");
9    }
10 }
```

（4行目）内容が2行なので波カッコは省略不可能
（7行目）1行しかないので波カッコは省略可能

ただし、実際の開発現場では、プログラミングのミスを防止するため、このような**ブロックの波カッコを省略することは推奨されない**のであわせて覚えておきましょう。

なぜ、波カッコの省略が推奨されないかは、章末の練習問題を解けばわかる。ぜひチャレンジしてほしい。

■ルール 2：ブロック内で宣言した変数の寿命

ブロックの中で、新たに変数を宣言することもできます。しかし、**ブロック内で宣言した変数は、そのブロックが終わると同時に消滅します**。たとえば、if 文のブロック内で宣言した変数は、そのブロックの外側では利用できません。このような「変数が利用可能な場所の範囲」のことを**スコープ**（scope）といいます。

図 3-5　ブロック内で宣言された変数のスコープは、そのブロック内に限られる

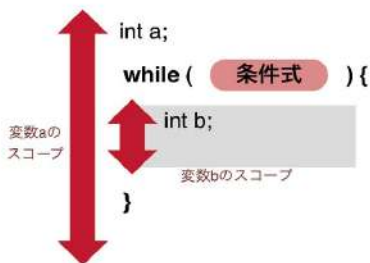



図 3-5 のようなコードで、while 文のブロックが終わった後で変数 b を利用しようとすると「シンボルを見つけられません。シンボル：変数 b」というコンパイルエラーになります。これは「変数 b を見つけられません」という意味です。

変数を宣言しているはずなのに「シンボルが見つかりません」とエラーが出たら、変数名のつづりとスコープを確認しよう。

3.3 条件式の書き方

3.3.1 条件式とは?

条件式とは、if文やwhile文で利用される式の一種で「処理を分岐する条件」や「繰り返しを続ける条件」を表現するためのものです(図3-6)。

条件式
if (tenki == true) {
もし変数tenkiの中身がtrueなら

条件式
while (age >18) {
もし変数ageが18より大きいなら

図3-6　条件式で「処理を分岐する条件」や「繰り返しを続ける条件」を表現する

ここで注目してほしいのは、条件式の中で用いられている"=="や">"といった記号です。これらの記号は**関係演算子**と呼ばれ、表3-1のような種類があります。

表3-1　関係演算子の種類と意味

演算子	意味
==	左辺と右辺が等しい
!=	左辺と右辺が異なる
>	左辺が右辺より大きい
<	左辺が右辺より小さい
>=	左辺が右辺より大きいか等しい
<=	左辺が右辺より小さいか等しい

関係演算子を使うと、たとえば、次のような条件式を考えることができます。

- **sw != false**　　**変数 sw が false でなかったら…**
- **deg - 273.15 < 0**　　**変数 deg から 273.15 を引いたものが 0 未満なら…**
- **initial == '雅'**　　**変数 initial に入っている文字が「雅」だったら…**

特に**「等しい」を表現する関係演算子はイコールが 2 つ "=="** であることに注意してください。誤ってイコールを 1 つ "=" しか書かないと代入演算子となり、まったく異なる動作になってしまいます。

うっかりイコール 1 つにしてしまいそうだから、気をつけなきゃ。

そうだね。これは初心者が犯すミスの代表格だよ。

条件式では == を使う

「条件式の中ではイコールは 2 つ」と覚えておきましょう。イコール 1 つの演算子 "=" を使うような条件式は、ほとんどありません。

3.3.2　if 文や while 文の正体

菅原さん、関係演算子も演算子なんですよね？　第 2 章で「演算子は評価されて別のものに化ける」と習ったのを思い出しましたが、「1 + 2」が「3」に化けるように、「age > 18」も別のものに化けるんですか？

そのとおりだよ。よく思い出したね。

第２章で学んだように、そもそも演算子とは「前後の値と一緒に評価され、別のものに化ける記号」のことでした。そして、先ほど学んだ関係演算子 "==" や ">" も、"+"（算術演算子）や "="（代入演算子）と同じ演算子の仲間であるため「評価されて化ける」という特性を持っています。具体的には**関係が成立するなら true（真）に、そうでないなら false（偽）に化ける**のです。

図 3-7　if 文では条件式を評価して、結果が true ならば、if 文以下の第 1 ブロックを実行する

したがって、if 文や while 文は次のように捉えることができます。

- **if 文とは「条件式の評価結果が true なら第 1 ブロックを、false なら第 2 ブロックを実行する」文。**
- **while とは「条件式の評価結果が true なら、ブロックを繰り返し実行する」文。**

なお、if 文や while 文は () の中の条件式の評価結果が true か false かで処理の流れを決定するため、「a + 3」のような算術演算子の式や、「b = 10」のような代入式を条件式に利用することはできません。なぜなら、これらの式は評価結果が true または false になる式ではないからです。

条件式のルール

if 文や while 文で用いる条件式は、評価結果が true または false になる式でなければならない。

3.3.3 文字列の比較

関係演算子の意味を理解できれば、条件式を記述することは難しいことではありません。しかし、「初心者のほぼ全員が必ず落ちる落とし穴」があります。

実は Java では、条件式の中で String 型の変数や文字列を比較する場合、**ある特別な書き方**をする必要があります。しかし初心者はついこのことを忘れてしまいがちです。

たとえば、「変数 s の内容が " 夕日 " という文字列だったら…」という条件式を考えてみましょう。ここまで学習した内容を踏まえれば、次のように書きたくなるでしょう。

```
if (s == "夕日") {
```

間違い！

一見、正しいように見えますが、Java のルールでは**文字列の比較は "==" ではできない**ことになっています。その理由については第Ⅲ部の「Java を支える標準クラス」で説明しますが、今のところは**文字列の比較を行う際には必ず次の書き方をする**と覚えてください。

```
if (s.equals("夕日")) {
```

正しい文字列の比較

文字列の比較

文字列型の変数 .equals(比較相手の文字列)

※「比較相手の文字列」としては、文字列のリテラルや変数を指定可。

※「文字列型の変数」と「比較相手の文字列」が等しい内容であれば、この式全体が true に化ける。

文字列が等しいか調べるときだけは "==" ではなくて "equals" なんですね。間違えないか不安です…。

そうだね。実は経験が豊富なプログラマでも、うっかり間違うことがある。しかも "==" を使ってもコンパイルエラーは起きず、「実行できるけどときどき変な動きをする」というタチの悪い不具合が起きるんだ。

3.3.4 論理演算子を用いた複雑な条件式

「年齢が 18 歳以上、かつ性別が男性」のように 2 つ以上の条件を組み合わせた、より複雑な条件式を使いたいことがあります。そのような場合には**論理演算子**を使います。

表 3-2　論理演算子の種類と意味

演算子	意味
&&	かつ（両方の条件が満たされた場合に true）
||	または（どちらか片方の条件さえ満たされれば true）

では実際に論理演算子を用いた例を見てみましょう。

```
if (age >= 18 && gender == 1) {…
if (name.equals("鈴木") || married == true) {…
```

図 3-8　&& では両方の条件が満たされなければ false になる。
|| ではどちらか片方の条件が満たされれば true になる

「&&」と「||」を組み合わせて、さらに複雑な条件式を作ることも可能です。以下の例は「age が 18 以上、かつ gender が 1」または、「age が 16 以上、かつ gender が 0」のときにブロックの中身が実行されます。

```
if ((age >= 18 && gender == 1) || (age >= 16 && gender == 0)) {…
```

朝香さんの結婚相手の条件って、複雑そうだね。

そうね。if((月給 > 私の月給 && 年齢 > 私の歳) || Java できる == true){ 結婚する ; } かしら。

なお「**もし〜でないならば**」のような否定形の条件式を作りたい場合は、条件式の前に**否定演算子 "!"** を付けます。

```
if (!(age == 10)) {
```
age が 10 に等しくない（10 以外）なら true

"!" は論理演算子の一種ですが、直後の条件式や値の true と false を逆転させる機能を持っています。

もしageが18以上で ないならば

if (!(age >= 18)){

評価

if (! true){

評価

if (false){

!は、その右にあるtrueやfalseを逆転させる演算子だよ

図 3-9　!は、それに続く評価式の結果を反転する役割を持つ

数学の表現と Java 条件式の表現

数学で「x は 10 より大きく、20 より小さい」という条件は「10 < x < 20」と表現します。しかし Java の条件式としては「10 < x && x < 20」と表記しますので注意してください。

おめでとう！ これで 2 人はブロックと条件式の両方をマスターし、どんな制御構造も書けるようになったよ。

それじゃ、この章の残りでは何を解説するんですか？

if 文や while 文をより便利に使うため、さまざまな書き方のバリエーションを紹介するよ。難しくないから気楽にいこう。

3.4 分岐構文のバリエーション

3.4.1　3種類の if 構文

if 文には 3 つのバリエーションがあります。最も基本的なものは、すでに紹介した if-else 構文です。さらに if のみの構文や、if-else if-else 構文があり、条件式の評価結果が false になった場合の流れが異なります。

図 3-10　if 文には 3 つのバリエーションがある

まずは基本形となる if-else 構文を確認しておきましょう。図 3-11 を見てください。この図は左から順に if-else 構文をフローチャートで表したもの、それを基本構文に置き換えたもの、そして Java のコードで表したものです。

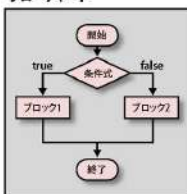



図 3-11　基本的な if-else 構文。条件式を評価し、true の場合にはブロック 1 を、false の場合にはブロック 2 を実行する

もし「**条件が満たされなかった場合、何もしない**」、すなわち、条件式の評価結果がfalseのときは何もしない（else直後のブロック内容が空）という場合はelseを省略できます。これが**ifのみの構文**です。

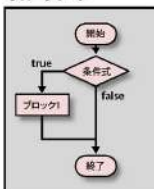



図3-12　if文のみの構文。条件式を評価し、trueの場合にはブロック1を、falseの場合には何もしない

一方、「**もし条件が満たされなかった場合、別の条件で評価したい**」ときには、else ifで始まるブロックをelseの前に挿入した「**if-else if-else構文**」を使用します。

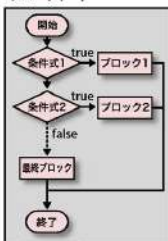



図3-13　複数の評価式で評価したい場合に用いるif-else if-else構文。else ifの数には限りはないので、いくつでも条件式を続けて記述できる

if-else if-else構文は、if-else構文（図3-11）や、ifのみの構文（図3-12）とは異なり、**1つのif文で3つ以上のルートに分岐できる**特徴を持っています。とても便利な構文なのですが、次のようなルールがあることを覚えておいてください。

if-else if-else 構文のルール

① else if ブロックは**複数記述できる**が、**if ブロックより後ろ、else ブロックより前にだけ**記述できる。
②最後の else ブロックは、中身が空ならば else ごと省略が可能。

「else if」って、「さっきの条件がだめなら、次の条件ならどう？」って、聞いているみたいですね。

3.4.2　switch 文による分岐

if-else if-else構文を使って「おみくじ」プログラムを書きました！ですが…冗長でスッキリしないんです。

リスト 3-4　冗長でスッキリしないソースコード

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("あなたの運勢を占います");
4      int fortune = new java.util.Random().nextInt(4) + 1;
5      if (fortune == 1) {
6        System.out.println("大吉");
7      } else if (fortune == 2) {
8        System.out.println("中吉");
9      } else if (fortune == 3) {
10       System.out.println("吉");
11     } else {
12       System.out.println("凶");
```

1～4の乱数発生

```
13          }
14      }
15  }
```

このコードには条件式が3つ含まれ、そのいずれもが「fortune ==(整数)」になっていることに注目してください。このように同じ変数に対して繰り返し比較を行っており、かつ次に挙げる2つの条件も満たす場合には、if 文をよりスマートな switch 文に書き換えることができます。

switch 文に書き換えることができる条件

①すべての条件式が「変数 == 値」や「変数 == 変数」のように左辺と右辺が**一致するかを比較する式**になっており、それ以外の「>」や「<」あるいは「!=」などが使われていない。

②比較する値が**整数**（byte 型、short 型、int 型のいずれか）、**文字列**（String 型）**または文字**（char 型）であり、小数や真偽値ではない。

では、switch 文の構文と、書き換え例を見てみましょう。

図 3-14 if 文を switch 文に置き換えたもの

switch 文を記述する場合には、以下の点に気をつけましょう。

switch 文記述の際の注意点

- switch の直後に書くのは、条件式（fortune == 1 など）ではなく、変数名（今回は fortune）。
- case（「～の場合は」という意味の英単語）の横には値を書き、その後ろにはコロン（:）を記述する（「case 値:」を case ラベル、または単にラベルと呼ぶ）。
- case 以降の処理の末尾には忘れずに break 文を記述する。
- default:（デフォルトラベル）は、それ以外の処理が不要な場合は省略可能である。

3.4.3 break 文を書き忘れると?

switch 文を利用するときに、特に注意しなければならないのが「**break 文の書き忘れ**」です。このミスを犯している占いプログラムがリスト 3-5 です。

リスト 3-5　break 文を忘れると…

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("あなたの運勢を占います");
4      int fortune = 1;                    常に 1
5      switch (fortune) {
6        case 1:
7          System.out.println("大吉");
8              ここに break; を入れ忘れている
9        case 2:
10         System.out.println("中吉");
```

```
11            break;
12          case 3:
13            System.out.println("吉");
14            break;
15          default:
16            System.out.println("凶");
17        }
18      }
19    }
```


実行結果

```
あなたの運勢を占います
大吉
中吉
```

このように7行目が実行され「大吉」と表示された後に、そのまま case 2 のブロックに進み、10行目も実行されてしまいました。

実は switch 文の正体は「**条件に一致する case ラベルまで処理をジャンプさせる命令**」にすぎません。そして **break 文で明示的に「処理を中断して switch 文を抜ける」**という指示がない限り、制御構造「順次」に従って次の case 文へ処理が進んでしまうのです。

いっそのこと、1つの case の範囲の実行が終わったら、break 文なしでも switch 文を抜けるような文法だったらよかったのになあ…。

確かにそう思うよね。でも「break 文をあえて書かないテクニック」もあるんだよ。

switch 文の「break 文がなければ次の case まで実行してしまう」という特徴を逆手にとって、次のような記述もできます。

リスト 3-6　あえて break 文を書かない

Main.java

```
public class Main {
  public static void main(String[] args) {
    System.out.println("あなたの運勢を占います");
    int fortune = new java.util.Random().nextInt(5) + 1;
    switch (fortune) {
      case 1:
      case 2:
        System.out.println("いいね！");
        break;
      case 3:
        System.out.println("普通です");
        break;
      case 4:
      case 5:
        System.out.println("うーん…");
    }
  }
}
```

1 ～ 5 の乱数発生

fortune が 1 か 2 なら…

fortune が 3 なら…

fortune が 4 か 5 なら…

条件式の短絡評価

Java は複雑な条件式を評価する際、少し賢い動作をします。たとえば図 3-8 左の条件式で変数 age の内容が 1 の場合、前半部分 (age>=18) を評価した時点で、条件式全体の結果が false になることが確定します。そのため、Java は続く後半部分 (gender==1) については無視して評価を行いません。このような Java のふるまいのことを、**短絡評価** (minimal evaluation) といいます。

3.5 繰り返し構文のバリエーション

3.5.1 2種類のwhile文

分岐のif文に3種類のバリエーションがあったように、繰り返しのwhile文にも2種類のバリエーションがあります。まずは、while文の基本形を復習しましょう。

while文では**ブロックを実行する前に条件式を評価**します（図3-15）。一方、do-while文を使えば、**ブロックを実行した後に条件式を評価**することができます（図3-16）。

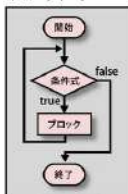



図3-15　while文の基本構文。まず条件式が評価される

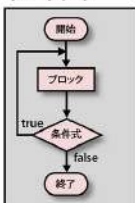



図3-16　do-while構文。ブロックを実行後に条件式が評価される

while 文と do-while 文が「条件式を判定するタイミング」が違うことはわかりましたが、結局どういう影響があるんですか？

ポイントは「ループの最低回数」だよ。

while 文はブロックを実行する前に条件判断を行う（これを「**前置判定**」といいます）ので、「初めから条件式の判定結果が false だった場合」は**1度もブロックが実行されません**。たとえば図 3-15 のサンプルコードで temp の内容が 10 だった場合、一度もブロックは実行されず、temp の内容も 10 のままです。

一方、do-while 文は、ブロックを実行後に条件判定を行う（これを「**後置判定**」といいます）ので、**最低1回はブロックを実行します**。図 3-16 のサンプルコードで temp が 10 だった場合でも 1 回はブロックが実行され、その結果 temp は 9 になります。

3.5.2 for 文による繰り返し

先輩。計算を「10 回繰り返したい」のですが、while 文を使っていたら、とてもわかりづらくなっちゃいました。

そうだね。では、回数指定でスマートにループを組むための新しい構文を覚えよう。

繰り返しの指定方法には「ある条件が成立するまで繰り返す」という方法と「～回繰り返す」という方法があります。そして後者のような回数指定の繰り返しは、while 文や do-while 文でも記述できないことはありませんが、**for 文**を使ったほうがスマートです。

まずは for 文を利用して「こんにちは」の表示を 10 回繰り返すプログラムを見てみましょう（リスト 3-7）。

リスト 3-7　基本的な for 文のサンプル

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     for (int i = 0; i < 10; i++) {
4       System.out.println("こんにちは");
5     }
6   }
7 }
```

なんだか for 文って難しそうですね。

決して難しくはないけれど、とっつきにくいね。まずは「基本形の丸暗記」から始めよう。

for 文の文法は少し複雑です。そこで文法の細かい解説は後で行うことにして、まずは「for 文の基本形」を見てみましょう。

図 3-17　for 文の基本形（10 回繰り返す例）

この基本形を完全に覚えてしまえば「100 回繰り返したい」「256 回繰り返したい」など、さまざまな場合に対応できます。なぜなら、この基本形の「10」の部分を「100」や「256」に変えるだけだからです。

3.5.3 for文の各部の意味

for文の基本形に慣れたら、いよいよfor文の詳細な構文を理解しましょう。for文に続くカッコの中は、図3-18のようにセミコロンによって区切られた3つの部分(①初期化処理、②繰り返し条件、③繰り返し時処理)から構成されています。それでは、それぞれの部分について詳しく見ていきましょう。

①初期化処理

forによる繰り返しが始まるにあたり**最初に1回だけ実行される文**です。通常、ここでは「何周目のループかを記録しておく変数」を定義します。このような変数を**ループ変数**といいます。

②繰り返し条件

ブロックの内容を実行する前に評価され、このループを継続するか否かを判定する条件式です。評価結果がtrueの間は"{"以降のブロックが繰り返し実行されます。なお、for文はwhile文と同じ「前置判定」の繰り返し構文であり、後置判定はできません。

③繰り返し時処理

for文内のブロックを最後まで処理して、**"}"まで到達した直後に自動的に実行される文**です。通常は、「i++」のようにループ変数の値を1だけ増やす文を書きます。

3.5.4 ループ変数

「①初期化処理」の中で宣言するループ変数って、普通の変数と同じものと考えていいんですか？

そうだね。いくつかの注意点を除けば普通の変数だよ。

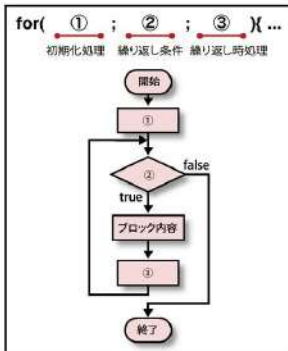
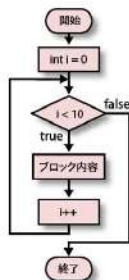



図 3-18 for 文の基本構文。3 つの部分は左から「初期化処理」「繰り返し条件」「繰り返し時の処理」で構成されている

ループ変数に関しては 3 つのポイントがあります。

ポイント 1:ループ変数の名前は自由

ループ変数の名前は、i に限らず自由に決めて構いません。ただし、for 文より前で、すでに宣言されている変数名は使えません。一般的には 1 文字程度の短い変数名が選ばれることが多いようです。

ポイント 2:ブロック内で利用可能

ループ変数も通常の変数の一種であるため、**ブロック内での計算や表示に使えます**。次の例ではループ変数 i の内容を表示しています。

リスト 3-8 for 文のループ変数 i の内容を表示する

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     for (int i = 0; i < 3; i++) {
4       System.out.print("現在" + (i + 1) + "周目→");
```

```
5        }
6    }
7 }
```

実行結果

現在1周目→現在2周目→現在3周目→

ポイント 3：ブロック外では利用不可能

ifブロック内で宣言した変数がブロック外では使えないように、ループ変数もfor 文のブロック内でのみ有効です。**for 文を抜けるとループ変数は消失してしまう**ので注意が必要です。

3.5.5 複雑な for 文

前節で学んだ「①初期化処理」「②繰り返し条件」「③繰り返し時処理」の 3 つを工夫することで、単に「～回繰り返す」という単純な繰り返しではなく、より高度な繰り返しを実現できます。以下に、for 文のさまざまなバリエーションの例を示します。

for (int i = 1 ; i < 10; i++) { …}	ループ変数を 1 からスタートする
for (int i = 0 ; i < 10; i+=2) {…}	ループ変数を 2 ずつ増やす
for (int i = 10 ; i > 0; i--) {…}	ループ変数を 10 から 1 ずつ 1 まで減らしていく
for (; i < 10 ; i++) {…}	ループ変数を初期化しない
for (int i = 0 ; i < 10 ;) {…}	繰り返し時の処理を行わない

3.6 制御構造の応用

3.6.1 制御構造のネスト

前節までで学んだ「分岐」や「繰り返し」の制御構造は、その中に別の制御構造を含むことができます。たとえば「分岐の中に分岐」や「繰り返しの中に分岐」などを入れることができ、このような多重構造を「**入れ子**」や「**ネスト**」といいます。

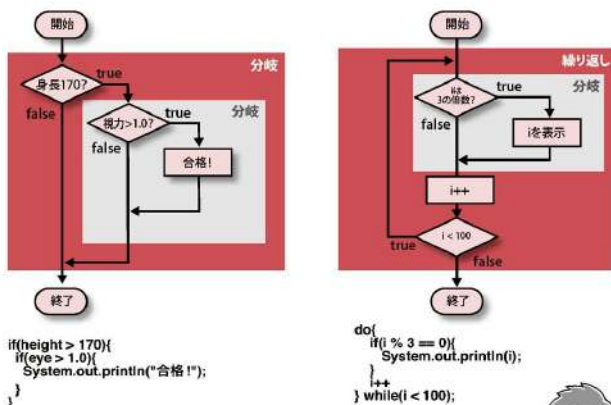



図 3-19　分岐処理の中の分岐処理（左側）と、繰り返しの中の分岐処理（右側）

それでは、for 文による繰り返しをネストさせて、「九九の表」を出力してみましょう。

リスト 3-9　for 文のループを 2 重にして九九の表を出力する

Main.java

```
1  public class Main {
```

```
2     public static void main(String[] args) {
3       for (int i = 1; i < 10; i++) {
4         for (int j = 1; j < 10; j++) {
5           System.out.print(i * j);  // 掛け算の結果を出力
6           System.out.print(" ");    // 空白を出力
7         }
8         System.out.println("");     // 改行を出力
9       }
10    }
11  }
```

iは1から9まで繰り返し

jも1から9まで繰り返し

実行結果

```
1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
⋮
```

内側のループが1周するたびに、掛け算の結果が空白を挟みながら右へと表示されていきます。内側のループが終了する（1つの段の掛け算の結果が出力される）と改行が出力されて、外側のループの1周が終了します。これを外側のループが終了するまで繰り返しています。

外側のループが1周目のときはiが1だから、内側のループでは「1×1、1×2…1×9」が、2周目のときはiが2だから「2×1、2×2…2×9」が実行されるのね。

3.6.2 繰り返しの中断

場合によってはfor文やwhile文を用いた繰り返しの途中で、その繰り返しを中断したいことがあります。このような場合のために、Javaでは**break文**と**continue文**という2種類の中断方法が用意されています。

図 3-20 break 文はループそのものを抜けて次のブロックを実行する。continue 文はループの「その周」だけを止めてループの先頭に戻り「次の周」のループを継続する

break 文は、break を囲んでいる最も内側の繰り返しブロックが即座に中断されるため、「while 文や for 文による繰り返しをすぐに中断したい」場合に利用します。一方の continue 文は「今の周回を中断して、**同じ繰り返しの次の周回に進む**」場合に利用します。

3.6.3 無限ループ

無限ループとは、強制停止されない限り**永久に繰り返しを続ける制御構造**のことです。プログラミングに慣れない間は、for 文や while 文の条件式を書き間違えて、意図せず無限ループに陥ってしまうことがあるため注意が必要です。

しかし、意図的に無限ループを作りたい場合もあります。無限ループを意図的に作るには、次の 2 つの記述方法が一般的です。

無限ループの作成方法

① while(true){…処理…}

② for(;;){…処理…}

3.7 第 3 章のまとめ

この章では、次のようなことを学びました。

制御構文

- 順次・分岐・繰り返しの 3 つの制御構造を組み合わせることで、どのようなプログラムも作成できる。
- 分岐と繰り返しは「条件式」と「ブロック」から構成されている。
- 条件式の評価結果は true または false でなければならない。
- 文字列を比較するときは「==」ではなく「equals」を使用する。
- ブロック内で定義した変数はブロック終了とともに消滅する。
- 制御構文はネストできる。

分岐

- if 文または switch 文を使用して分岐を実現する。
- if 文は「if のみの構文」「if-else 構文」「if-else if-else 構文」の 3 種類。
- switch 文のブロックは break 文で抜けることができる。

繰り返し

- while 文、do-while 文または for 文を使用して繰り返しを実現できる。
- while 文のブロックは最低 0 回以上、do-while 文のブロックは最低 1 回以上実行される。
- for 文はループ変数を用いて「～回繰り返す」という場面に使用する。
- break 文を実行すると繰り返し自体を中断し、continue 文を実行すると繰り返しの次の周回へ進むことができる。

3.8 練習問題

練習 3-1

次の日本語で記載された条件式を Java で記述してみましょう。

①変数 weight の値が 60 に等しい。
②変数 age1 と age2 の合計を 2 倍したものが 60 を越えている。
③変数 age が奇数である
④変数 name の中身の文字列が「湊」と等しい。

練習 3-2

次に挙げる A から F の式のうち、条件式として利用できるものを選んでください。

A. cost = 300 * 1.05
B. 3
C. age != 30
D. true
E. b + 5 < 20
F. gender = true

練習 3-3

次の内容に沿った Java プログラムを作成してください。

① int 型の変数 seibetsu を定義し、0 か 1 を代入する（どちらでも構わない）。また、int 型変数 age を定義し、適当な数字を代入する。
② 画面に「こんにちは。」と表示する。
③ もし変数 seibetsu が 0 であれば「私は男です。」、そうでなければ「私は女です。」と表示する。
④ もし変数 seibetu 男なら age 変数の中身を表示し、続けて「歳です。」と表示する。
⑤最後に「よろしくおねがいします。」と表示する。

練習 3-4

次のようなjavaコードがあります。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      boolean tenki = true;
4      if (tenki == true) {
5        System.out.println("洗濯をします");
6        System.out.println("散歩にいきます");
7      } else
8        System.out.println("DVDを見ます");
9    }
10 }
```

あなたは3行目のtenkiがfalseの場合、「DVDを見ます」の後に「寝ます」を表示するように変更するため、8行目の次に「寝ます」と表示する行を以下のように追加しました。

```
6        System.out.println("散歩にいきます");
7      } else
8        System.out.println("DVDを見ます");
9        System.out.println("寝ます");
10   }
11 }
```

この行を追加した

しかし、このプログラムは意図したように動きません。どの部分に誤りがあり、どのような現象が発生しているかを答えてください。そして、誤りを修正するには、どうすればよいかを考えてください。

練習 3-5

switch 文を用いて次の条件を満たすプログラムを作成してください。

① 画面に「[メニュー] 1:検索 2:登録 3:削除 4:変更 >」と表示する。
② 数字を入力し、変数 selected に代入する（ヒント:2.6.6 項を参照）。
③ もし変数 selected が 1 なら「検索します」、2 なら「登録します」、3 なら「削除します」、4 なら「変更します」と表示する。
④ selected が 1 から 4 のいずれでもない場合は何もしない。

練習 3-6

次の条件に従ってプログラムを記述してください。

① 画面に「【数あてゲーム】」と表示する。
② 0 から 9 までの整数の中からランダムな数を 1 つ生成して変数 ans に代入する（ヒント:2.6.5 項を参照）。
③ for 文を用いた「5 回繰り返すループ」を作る。その際に以下の④～⑦は、ループの中に記述する。
④ 画面に「0 ～ 9 の数字を入力してください」と表示する。
⑤ 数字を入力し、変数 num に代入する（ヒント:2.6.6 項を参照）。
⑥ もし変数 num が変数 ans と等しかったら「アタリ！」と画面に表示して繰り返しを終了する。
⑦ もし変数 num が変数 ans と等しくない場合は「違います。」と表示する。
⑧ 繰り返しのブロックの外側で、「ゲームを終了します」と画面に表示する。

3.9 練習問題の解答

練習 3-1 の解答

① weight == 60　② (age1 + age2) * 2 > 60

③ age % 2 == 1　④ name.equals(" 湊 ")

練習 3-2 の解答

条件式として適切なものは C、D、E です。

練習 3-3 の解答

Main.java

```
public class Main {
  public static void main(String[] args) {
    int seibetsu = 0;
    int age = 52;
    System.out.println("こんにちは。");
    if (seibetsu == 0) {
      System.out.println("私は男です。");
    } else {
      System.out.println("私は女です。");
    }
    if (seibetsu == 0) {
      System.out.println(age + "歳です。");
    }
    System.out.println("よろしくおねがいします。");
  }
}
```

練習 3-4 の解答

不具合の原因は、elseの後に波カッコ"{"と"}"がなく、else文のブロックが「DVDを見ます」で終了しているためです(その次の「寝ます」の行はelse文のブロックに含まれていない)。

「雨が降っても晴れていても『寝ます』が表示されてしまう」という現象が発生しています。それを修正するには、次のリストのように「DVDを見ます」「寝ます」を表示する2行を波カッコで囲みます。

```
6        System.out.println("散歩にいきます");
7      } else {
8        System.out.println("DVDを見ます");
9        System.out.println("寝ます");
10     }
11   }
12 }
```

練習 3-5 の解答

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.print
           ("［メニュー］　1：検索　2：登録　3：削除　4：変更>");
4      int selected = new java.util.Scanner(System.in).nextInt();
5      switch (selected) {
6        case 1:
7          System.out.println("検索します。");
8          break;
9        case 2:
10         System.out.println("登録します。");
11         break;
12       case 3:
13         System.out.println("削除します。");
```

```
14          break;
15        case 4:
16          System.out.println("変更します。");
17          break;
18      }
19    }
20  }
```

練習 3-6 の解答

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.print("【数あてゲーム】");
4      int ans = new java.util.Random().nextInt(10);
5      for (int i = 0; i < 5; i++) {
6        System.out.println("0〜9の数字を入力してください");
7        int num = new java.util.Scanner(System.in).nextInt();
8        if (ans == num) {
9          System.out.println("アタリ！");
10         break;
11       } else {
12         System.out.println("違います。");
13       }
14     }
15     System.out.println("ゲームを終了します");
16   }
17 }
```

第 4 章

配列

第 1 章では、さまざまな数や文字列を入れることができる
変数のしくみを学びました。
もちろん、作成するプログラムが大きくなると、
たくさんの変数を使うことになります。
たとえば、複数の変数をすべて足したり、
平均を求めたりするようなこともあるでしょう。
この章で学ぶ「配列」は、変数をより便利に使うためのしくみです。
配列を使うことで、一度に多くの変数を処理することができます。

CONTENTS

配列のメリット

4.1.1 変数が持つ不便さ

湊くん、何を作っているのかな？

第1章で学んだ変数を使って、テストの点数の合計や平均を算出するプログラムを作っています。我ながらよくできていると思います。

よくできているね。でも、配列を使うとよりよくなるよ。

え～！ せっかく作ったのに…。

配列をマスターしたら、プログラムが楽になるよ。便利だから、しっかり身に付けよう。

第1章では数字や文字列などのデータを格納して扱う変数のしくみを学びました。そして第3章では、変数を用いることで分岐や繰り返しを実現できることがわかりましたね。このように、変数だけでもプログラムを書くことはできます。しかし、それだけでは少し不便なこともあります。湊くんが作成したテストの点数を管理するプログラムを見て考えてみましょう。

リスト 4-1　点数管理プログラム

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
```

```
3        int sansu = 20;
4        int kokugo = 30;
5        int rika = 40;
6        int eigo = 50;
7        int syakai = 80;
8
9        int sum = sansu + kokugo + rika + syakai + eigo;
10
11       int avg = sum / 5;
12       System.out.println("合計点:" + sum);
13       System.out.println("平均点:" + avg);
14   }
15 }
```


一見、問題なさそうに見えますが、このコードには不便なことが 2 つあります。

■科目が増えるたびに、それをコードに追加しなければならない

新しい科目を加えるには、taiiku（体育）などの変数を宣言した後に、合計と平均を算出している行を修正する必要があります。これを忘れると新しい科目の点数が追加されなくなってしまいます。

■まとめて処理できない

「点数の良い科目から順に並び替える」などの処理を行う場合、コードが長く、複雑になってしまいます。

これらの原因は、コンピュータが 5 つの科目の変数を「個々の独立したデータ」として扱っていることにあります。

私たち人間は、この 5 つの変数は「各科目のテストの点数を格納している変数で、**1 組のもの**として計算（合計の算出など）することがある」と無意識に考えています。しかし、コンピュータにとって変数は「何の関係もないバラバラの 5 つの箱」でしかなく、1 組のものとして扱うことができないのです。

複数のデータを 1 組のものとして扱うことは、私たちの身の回りに多くあります。

例として携帯電話のアドレス帳を考えてみましょう。アドレス帳に鈴木さんのデータがあるとした場合、そこには鈴木さんの「名前」「電話番号」「メールアドレス」など複数の情報を「鈴木さんの情報」として1組で管理しています。そして、アドレス帳には鈴木さんのほかにも多くの友人・知人のデータが格納されており、それらは1人1組ごとにコピーや削除が可能です。

このように、現実世界では1つのグループに属するデータをまとめて扱うことが多くあります。それにもかかわらず、これをプログラムで実現できないとなると、前ページに挙げたような不便が発生してしまいます。

そこで、ほとんどのプログラミング言語では、「いくつかの関係あるデータをグループにして、まとめて1つの変数に入れる」しくみが用意されています。1つの変数に複数のデータを入れるといっても、雑多に放り込むわけではなく、きちんとした構造に整理して、後から特定の値を取り出せるように格納することができます。

この構造のことを**データ構造**（data structure）といい、その代表的なものが本章で学ぶ「**配列**」です。

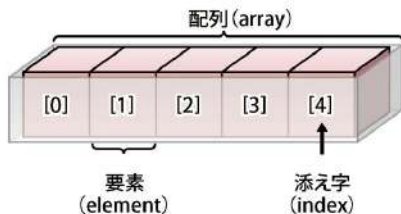
データ構造は、複数のデータをひとまとめにする方法で、何種類かあるということですね。そして、その1つが「配列」ということですか？

そのとおりだよ。配列以外にも「マップ」や「スタック」などのデータ構造もあるんだ。

4.1.2 配列とは

配列（array）とは、**同一種類の複数データを並び順で格納するデータ構造**です。右の概念図を見てください。

図 4-1　配列の構造

変数のような箱が連続して並んでいます。そして、この箱の1つ

ひとつを**要素**(element)といいます。要素は変数と基本的に同じで、型を持ち、データを格納できます。また、配列に含まれる各要素の型は揃える必要があります。ある要素の型がint型ならば、その他の要素の型もint型でなければいけません。よって**配列の各要素には同一種のデータしか格納できない**ということになります。「要素の0番には数字、要素の1番には文字列」などのような異なるデータ型は格納できません。

これにより、配列ではそれぞれの要素に値を代入して、使用することができますし、すべての要素のデータをひとまとめに扱うこともできます。そのため、「配列のすべての要素を合計せよ」、「配列のすべての要素の中身を大きい順に並び替えよ」といったことが、変数よりも簡単に行うことができます。

図4-1にあるように配列内の各要素には0番、1番、2番という番号が付いています。この要素の番号を**添え字**(index)といい、**0から始まる決まり**になっています(1から始まるのではないことに注意してください)。たとえば、要素が5つあったら0番の要素、1番の要素…4番の要素と表現されます。このとき5番という添え字の要素はありません。

配列の要素は0から始まる!

最初の要素が0番であることは、初学者はもちろん、熟練者でもうっかり間違えることがあります。しっかり覚えておきましょう。

配列を使うには、どうすればいいんですか?

変数と同じで、使う前に準備が必要なんだ。ただし、変数より少しだけ複雑な準備が必要だから、次節でしっかり紹介しよう。

4.2 配列の書き方

4.2.1 配列の作成

配列を作成するには、以下の 2 ステップが必要です。

Step1　配列変数の宣言
Step2　要素の作成と代入

まず、Step1 で**配列変数**を作成します。この変数は今までの変数と異なり、代入できるのは値ではなく要素です。そして Step2 で要素を作成し、それを Step1 で作成した配列変数に代入することで配列が完成します。

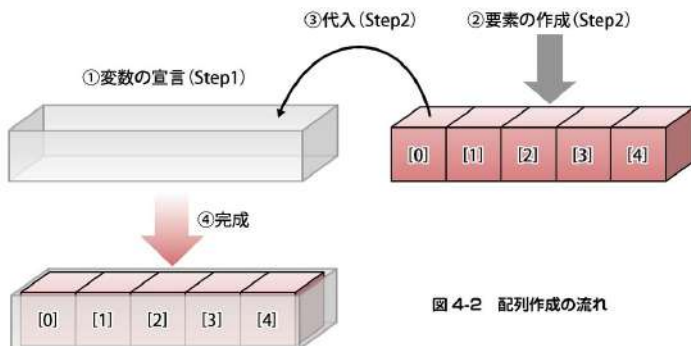



図 4-2　配列作成の流れ

Step1　配列変数の宣言

配列変数を作成するには、今まで同様に宣言が必要です。型を指定するには、要素の型の後に [] を付けて記述します。たとえば、int 型の要素を代入する配列の場合、次のように宣言します。

```
int[] score;
```

この宣言で使われている「int[]」型は、見た目は int 型と似ていますがまったく異なる型なので注意してください。詳しくは後で解説しますので、ここでは「int 型の要素を代入できる型」だと理解してください。

代入する要素が double 型の場合は「double[]」で、String のときは「String[]」なんですね。

配列変数の作成（宣言）

要素の型 []　配列変数名

Step2　要素の作成と代入

次に要素を作成して Step1 で宣言した配列変数に代入します。たとえば「int 型の要素を 5 個作り、配列変数 score に代入する」には、次のようにします。

```
score = new int[5];
```

new は「new 演算子」と呼ばれるもので、指定された型の要素を [] 内の数値の分だけ作成します。作成された要素は、「=(代入演算子)」で配列変数に代入することができます。

うーん。配列は変数と違って、手順がややこしいなあ。

そうよね。ここまでの内容をまとめてみましょうか。

リスト 4-2　配列の作成手順

Main.java

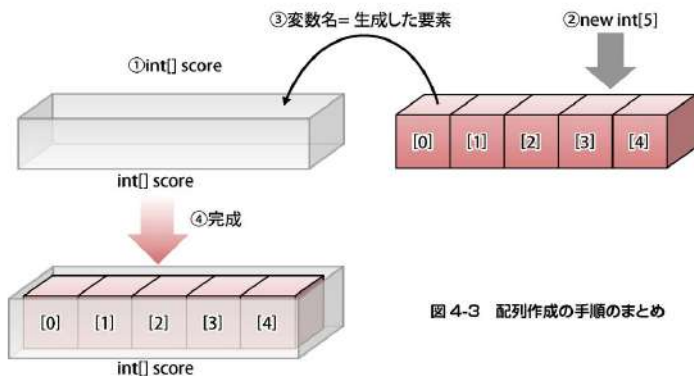
```
1  public class Main {
2    public static void main(String[] args) {
3      int[] score;
4      score = new int[5];
5    }
6  }
```


図 4-3　配列作成の手順のまとめ

　また、次のようにして先ほどの「Step1 の配列変数の宣言」と「Step2 の要素の作成と代入」を同時に行うことができます。

リスト 4-3　配列の作成手順（Step1 と Step2 を同時に行う）

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int[] score = new int[5];
4    }
5  }
```

なお、いくつ要素を作ったか（配列の要素数はいくつか）は自分で覚えておかなくても大丈夫です。次のようにして「配列名 .length」で調べることができます。

リスト 4-4　配列の長さを調べる

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int[] score = new int[5];
4     int kobakos = score.length;          5になる
5     System.out.println("要素の数: " + kobakos);
6   }
7 }
```

点数管理のプログラムで、平均を算出するときに使うといいわよ。

なるほど！　そうすれば科目の数が増えても、平均の算出の部分は修正しなくてもいいね。

配列の要素数の取得

配列変数名 .length

4.2.2 配列の利用方法

配列に含まれるそれぞれの要素は変数と同じように扱えますが、どの要素に値を出し入れするかを指定するため、「score[1] = 10;」のように添え字を用いて利用します。ここで「**配列の最初の要素は 0 番である**」というルールを思い出してください。「score[1] = 10;」は、配列 score の**先頭ではなく、先頭から 2 番目の要素**に 10 を代入していることになります。

次のリストを見てください。これは score の 2 番目の要素に 30 を代入し、それを 6 行目で表示しています。

リスト 4-5　配列の要素に値を代入

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int[] score;
4     score = new int[5];
5     score[1] = 30;
6     System.out.println(score[1]);
7   }
8 }
```

要素 score[1] に代入

要素 score[1] の中身を表示

4.2.3 配列の初期化

ところで、変数の値を取り出す前には、必ず値を代入して初期化しなければなりません。初期化をしていない変数を利用するとコンパイルエラーが発生します。

リスト 4-6　初期化されていない変数を利用

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int x;
4     System.out.println(x);
5   }
6 }
```

x が初期化されていないので、コンパイルエラーになる

しかし、**配列の要素は自動的に初期化されます**。たとえば、次のように int 型の要素を持つ配列を用意した場合、5 つの要素はすべて 0 で初期化されます。

リスト 4-7　配列の初期化

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int[] score = new int[5];
4     System.out.println(score[0]);
5   }
6 }
```

0が出力される（エラーにならない）

要素がどのような値で初期化されるかは、要素の型によって決められています。

int や double 等の数値の型	0
boolean 型	false

なお、String 型の要素は後ほど学習する「**null**」という値で初期化されます。

4.2.4 省略記法

これまで見てきた配列の作成と初期値の代入を同時に行うことができます。

配列作成と初期化の省略記法

① 要素の型 []　配列変数名 = new 要素の型 []{ 初期値 1, 初期値 2, 初期値 3,…};

② 要素の型 []　配列変数名 = { 初期値 1, 初期値 2, 初期値 3,…};

省略記法の具体例を次に示します。

```
int[] score1 = new int[] { 20, 30, 40, 50, 80 };
int[] score2 = { 20, 30, 40, 50, 80 };
```

省略記法①
省略記法②

4.3.1 範囲外要素の利用による例外の発生

先輩、コンパイルできて実行もできたんですけど、英文が表示されて止まっちゃいました。これって何ですか？

これは例外が発生したんだね。配列の使い方を間違うと表示されるエラーメッセージだよ。

リスト 4-8 点数管理プログラム（配列版）

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int[] score = { 20, 30, 40, 50, 80 };
4      int sum =                                              ← 合計の算出
         score[1] + score[2] + score[3] + score[4] + score[5];
5      int avg = sum / score.length;                          ← 平均の算出
6      System.out.println("合計点:" + sum);                   ┐
7      System.out.println("平均点:" + avg);                   ┘ 合計と平均の表示
8    }
9  }
```

実行結果（エラー）

```
java.lang.ArrayIndexOutOfBoundsException:5 Main.java(4)
⋮
⋮
```

湊くんが、どこで間違えたかわかりましたか？ 配列 score の要素数は 5 つなので要素の添え字は [0] ～ [4] ですね。しかし、このコードは 4 行目で score[5] を使ってしまっています。

このように、存在しない要素をコード内で使っていてもコンパイルは成功します。しかし、プログラムを動かすと、その行を処理しようとした際に **ArrayIndexOutOfBoundsException** というエラーメッセージが表示されプログラムが中断してしまいます。このようなエラーを、特に**例外**（exception：エクセプション）といいます。

例外については、詳しくは第 15 章で解説します。現時点では、実行中に「～Exception」と表示されて中断したら、例外というエラーが発生したと判断できるようにしておきましょう。

あれほど、先頭の要素の添え字は [0] だと注意されていたのに…。

これは経験者でも、うっかりやってしまうミスだよ。ArrayIndexOutOfBoundsException が発生したら、「エラーの原因は存在しない要素を使おうとしたからだ」と判断しよう。

Array は配列、Index は添え字、OutOfBounds は範囲外という意味で覚えればラクですね。

英語が表示されたからといって慌てる必要はない。どれも簡単な単語だ。逃げずに読むことが実は上達への近道だよ。

4.4 配列のデータをまとめて扱う

4.4.1 for文と配列

よし！ 例外が出なくなったぞ。これで完璧だ！！

最初に比べてよくなったね。でも、まだ改善できる箇所があるよ。今のままだと、将来科目が増えた場合に合計点を算出する箇所に修正が必要だろう？　そのめんどうも解決できるんだよ。

実は、配列の添え字の指定に変数を用いることができます。たとえば、変数aに3が入っているとき、「score[a]」という記述をすれば、前から4番目の要素にアクセスできるのです。

このことを利用して、次のリスト4-9のようなことができます。

リスト4-9　配列とfor文

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int[] score = { 20, 30, 40, 50, 80 };
4      for (int i = 0; i < score.length; i++) {
5        System.out.println(score[i]);
6      }
7    }
8  }
```

score[0],score[1]…

変数iに注目してください。forループにより、iは、0→1→2→3→4と

変化していくので、score の添え字に変数 i を用いることで score[0] から score[4] を表示させることができます。このように最初の要素から最後の要素まで、代入されている値を順番に使用することを「配列を回す」ともいいます。

そして、3 行目で作成している要素の数が 6 つや 7 つに変わっても、この「全内容の表示部分」はいっさい修正が不要な点に注目してください。**for 文の終了条件に配列変数名 .length を用いる**ところがポイントです。

このように for 文を用いて配列の各要素の値を順番に取り出し、それに対して何らかの処理を行うコードを書く機会は多いので、必ず身に付けておきましょう。

配列を for ループで回す

```
for ( int i = 0 ; i < 配列変数名 .length ; i++ ) {
            ⋮
}
```

4.4.2 拡張 for 文

拡張 for 文は、配列の要素を 1 つずつ取り出すループを簡単に書くために導入された新しい for 文の文法です。

拡張 for 文で配列を回す

```
for ( 要素の型　任意の変数名 : 配列変数名 ) {
            ⋮
}
```

拡張 for 文では、ループが 1 周するたびに次の要素の内容が「任意の変数名」で指定した変数に格納されるため、ブロックの中ではその変数を利用して要素を取り出します。従来の for 文と比較しながら次のコードを見てください。

リスト 4-10　従来の for 文での例

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int[] score = { 20, 30, 40, 50, 80 };
4     for (int i = 0; i < score.length; i++) {
5       System.out.println(score[i]);
6     }
7   }
8 }
```

配列名 .length で要素数を得る

要素を指定して取り出す必要がある

リスト 4-11　拡張 for 文の例

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int[] score = { 20, 30, 40, 50, 80 };
4     for (int value : score) {
5       System.out.println(value);
6     }
7   }
8 }
```

ループが 1 周するたびに次の要素が value に入る

value をそのまま使える

拡張 for 文を使うと、コードにループ変数や添え字などが登場しなくなり、すっきりします。

4.5 配列の舞台裏

4.5.1 配列を理解する

配列も回せるようになったし、これで配列は完璧ですね！

確かに基本的な内容は十分マスターしたね。でも、まだ知っておくべきことがあるよ。次のコードを見てごらん。最後に何が出力されると思う？

リスト 4-12　実行結果は？

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     int[] a = { 1, 2, 3 };
4     int[] b;
5     b = a;
6     b[0] = 100;
7     System.out.println(a[0]);
8   }
9 }
```

a[0] は 1

b[0] に 100 を代入

a[0] を表示すると…

「1」が出力されると思いましたか？ しかし、実際に出力されるのは「100」なのです。

なぜ「100」が表示されるかを理解するために、この節では「配列を利用しているとき、コンピュータの中では何が起きているか」という舞台裏を解説します。この舞台裏を十分理解できているかどうかで、今後の章における理解や応用力に格段の差が付きます。やや高度な内容ですが、ぜひ理解してください。

4.5.2 メモリと変数

これまで p.142 の図 4-2 のように、配列は「～ [] 型の配列変数に～型の要素を入れて作成する」というイメージを示してきました。実は、これは私たち人間に理解しやすくするためのイメージ図であって、実際にコンピュータの中でこのような構造になっているわけではありません。

コンピュータは使用するデータをメモリに記録します。メモリの中は碁盤の目のように区画整理されており、各区画には住所（アドレス）が振られています。そして変数を宣言すると、空いている区画（どこが選ばれるかわからない）を変数のために確保します（変数の型によって何区画を使用するかは異なります）。変数に値を代入するとは、確保しておいた区画に値を記録することなのです。

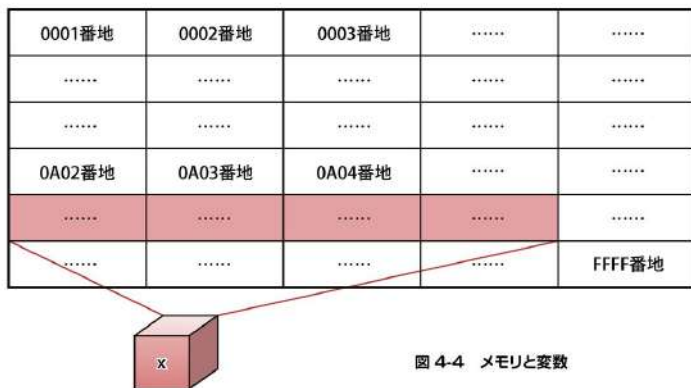
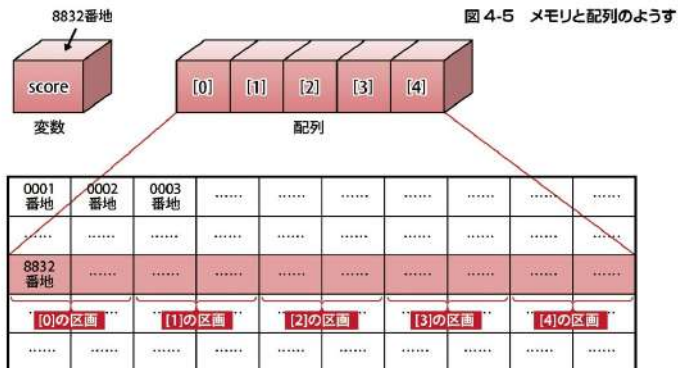



図 4-4　メモリと変数

4.5.3 メモリと配列

では、配列を作成（「int [] score = new int[5]; 」）したとき、メモリの中ではどのようなことが起きているのでしょうか。

配列変数の宣言により int[] 型変数が、new 演算子により配列の実体（要素の集まり）がメモリ上の区画に作成されます（図 4-5）。そして、配列変数には 5 つ

図 4-5　メモリと配列のようす

の要素まるごとではなく、**「最初の要素のアドレス」が代入されます。**

int[] score = new int[5]; を実行したときのメモリ上のようす

① int 型の要素を 5 つ持つ配列がメモリ上に作成される。
② int[] 型の配列変数 score がメモリ上に作成される。
③ score に配列の先頭要素のアドレスが代入される。

int[] 型だけでなく、double[] 型や String[] 型の配列変数も配列の先頭要素のアドレスを代入するんですね。

そうだよ。ただし、int[] 型の配列変数に、要素が double 型である配列の先頭要素のアドレス は代入できない。int[] 型の配列変数は、要素が int 型である配列の先頭要素のアドレスしか代入できないんだ。

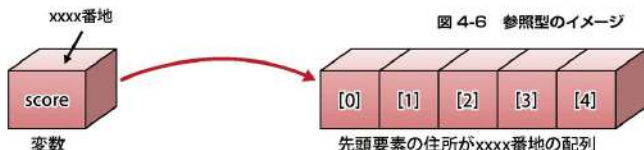
Java は次のようなしくみで配列の要素を利用しています。

プログラムから score[n] と指定されたら

① score の中に入っている番地 (=8832) を取り出し、配列（先頭要素）を見つける。

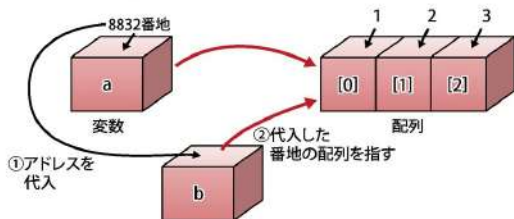
②見つけた配列の先頭要素から n 個後ろの要素の区画を読み書きする。

配列変数 score は、「配列の実体は 8832 番地にあります」と指し示す動作をしていることになります。このことを「**参照**」と呼びます。また、具体的なデータではなく、メモリ上の番地を代入する変数のことを「**参照型**」（reference type）変数といい、int や boolean のような「基本型」変数と区別します。

図 4-6　参照型のイメージ

4.5.4　配列を複数の変数で参照

ここまでの内容を十分に理解すれば、先ほどのリスト 4-12 ではどのような状況になっているかが想像できるでしょう。

変数aに番地「8832」が入っているとした場合、5行目でコピーされているのは、この 8832 になります。その結果、**変数 b は変数 a と同じ配列を参照する**ことになります。

図 4-7　a と b とが同じ配列を参照している

よって、a[0] と b[0] は、まったく同一のものを指していることになります。この状態で「b[0] = 100;」を実行することは「a[0]=100;」と同じです。当然、7 行目では 100 が出力されます。

4.6 配列の後片付け

4.6.1 ガベージコレクション

次のコードを見てください。

リスト 4-13 ガベージコレクション

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     boolean b = true;
4     if (b == true) {
5       int[] i = { 1, 2, 3 };
6     }
7   }
8 }
```

if ブロック内で配列を作成

5 行目で、配列変数 i が宣言され、同時に 3 つの要素を持つ配列が生成されています。しかし、変数の寿命は自分が宣言されたブロックが終了するまで（3.2.1項）だったことを思い出してください。これは配列変数でも同じです。

つまり、6 行目の時点で配列変数 i はメモリから消滅します。一方、**new で確保された要素たちは普通の変数ではありませんので、ブロックが終了しても寿命は迎えません。**その結果、配列はどの配列変数からも参照されない状態でメモリに残ってしまいます。

残った配列は、Java のプログラムからどのような方法を使っても読み書きすることはできず、事実上メモリ内のゴミ（garbage）となります。ゴミとなってしまった配列を放置し続けると、こういったゴミが溜まり続け、メモリを圧迫してしまう可能性があります。

本来ならば、「使用できなくなった配列は、もう使いませんから、破棄して（区

画から取り除いて）メモリ領域をお返しします」という**メモリの後片付けをプログラマが行わなければなりません。**

しかし、Java には**ガベージコレクション**（GC: garbage collection）というしくみが常に動いており、実行中のプログラムが生み出したメモリ上のゴミ（＝どの変数からも参照されなくなったメモリ領域）を自動的に探し出して片付けてくれます。

勝手にゴミを探して片付けてくれるなんて自動掃除機みたい。楽でいいわね♪

4.6.2 null

先の項では変数の寿命によって配列変数が配列を参照しなくなる例でしたが、**null** を使用することで、意図的に参照されないようにすることができます。次のコードを見てください。

リスト 4-14　null の利用

Main.java

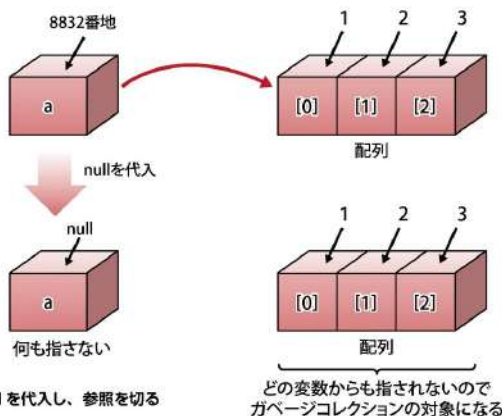
```
1 public class Main {
2   public static void main(String[] args) {
3     int[] a = { 1, 2, 3 };
4     a = null;
5     a[0] = 10;
6   }
7 }
```

配列変数 a に null を代入

上記の 4 行目では配列変数に null という特別な値を代入しています。null とは「何もない」という意味で、配列変数 a のような参照型の変数に代入することができます。null が代入されると、参照型の変数はどこも参照していない状態になります。このように、ある番地を参照していた配列変数に null を代入し、参照させなくすることを「参照を切る」ともいいます。

null とは？

① int[] 型などの参照型変数に代入すると、その変数は何も参照しなくなる。
② int 型などの基本型変数には代入することができない。

図 4-8　null を代入し、参照を切る

4.6.3 NullPointerException

先輩、さっきのプログラム、4 行目で null を代入した後、5 行目で「 a[0] = 10; 」していますけど、a はどこも参照していませんよね？

よく気づいたね。でも、コンパイルは成功するんだ。実行するとどうなるかな？

あっ、またエラーになりましたよ！ NullPointerException…例外ですね！

リスト 4-14 を実行すると、次のようなエラーが画面に表示されます。

実行結果（エラー）

```
Exception in thread "main" java.lang.NullPointerException
  at Main.main(Main.java:5)
```

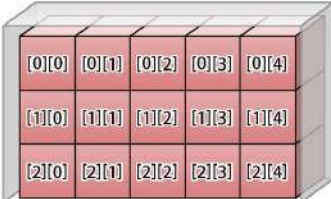
この例外は、null が格納されている配列変数を利用しようとすると発生します。先ほど学習した ArrayIndexOutOfBoundsException とともに、配列関連で見かけることの多い例外です。

配列の length と文字列の length()

この章では、配列の要素数を取得するために「配列変数名.length」という記述が可能なことを学びましたが、これと似た記述方法として、「文字列変数名.length()」があります。

String 型の変数の後に「.length()」を付けることで、その文字列型変数に格納されている文字列の長さ（文字数）を得ることができます。その際には、全角／半角を問わず 1 文字としてカウントされますので、たとえば次のコードを実行すると、画面には 7 が表示されます。

```
String s = "Javaで開発";
System.out.println(s.length());
```

配列の「length」と文字列型の「length()」は、どちらも長さを取得するためのものですが、「文字列のときは () を付ける、配列のときは () を付けない」と覚えておきましょう。

4.7 多次元の配列

4.7.1 多次元配列とは?

今まで学習してきた配列は1次元配列といいます。1次元配列に縦の並びを加えると**2次元配列**になります。

2次元配列は図4-9のように要素が縦横に並んだ表のようなものです。データを表のような形で扱いたいときに使用すると便利です。

図4-9　2次元配列のイメージ

ちなみに、2次元以上の配列を多次元配列と呼びます。ビジネスアプリケーションの開発では多次元配列を使う機会は少ないですが、科学技術計算などでは多く利用されます。

2次元配列の宣言

要素の型 [][]　配列変数名 = new 要素の型 [行数][列数];

2次元配列の要素の利用

配列変数名 [行の添え字][列の添え字]

たとえば、兄弟2人の3科目のテスト結果を格納する2次元配列は、次のように書くことができます。

リスト 4-15　2 次元配列

Main.java

```
public class Main {
  public static void main(String[] args) {
    int[][] scores = new int[2][3];
    scores[0][0] = 40;
    scores[0][1] = 50;
    scores[0][2] = 60;
    scores[1][0] = 80;
    scores[1][1] = 60;
    scores[1][2] = 70;
    System.out.println(scores[1][1]);
  }
}
```

2 行 3 列の配列

このリストの 2 次元配列を図にすると下のようなイメージです。

2 次元配列では 1 次元配列と異なり [] を 2 つ使用します。最初の [] で行、次の [] で列を指定します。

図 4-10　2 行×3 列の点数表

しかし、図 4-10 はあくまでもイメージです。Java における 2 次元配列は、正確には「表」ではなく、「配列の配列」になります。まず、先ほどの「表」のイメージを「配列の配列」というイメージに変化させてみましょう。2 行× 3 列の表の場合、要素数 2 の行配列（親配列）の中に、それぞれ要素数 3 の列配列（子配列）が入ります。実際のメモリ上では、図 4-11 のようになっています。

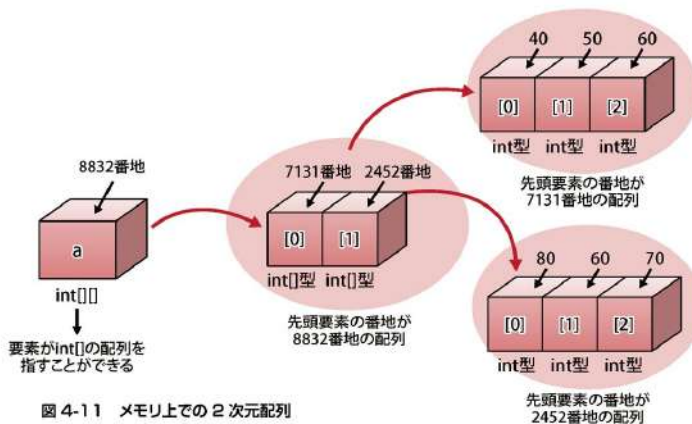



図 4-11　メモリ上での 2 次元配列

この図 4-11 の状態は次のプログラムで確認できます。

リスト 4-16　親配列と子配列の要素数を表示

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int[][] scores = { { 10, 20, 30 }, { 30, 40, 50 } };
4      System.out.println(scores.length);
5      System.out.println(scores[0].length);
6    }
7  }
```

このような初期化が可能（3行目）
2 が出力される（4行目）
3 が出力される（5行目）

第4章のまとめ

この章では、次のようなことを学びました。

配列の基礎

- 配列とは、同じ型の複数の値をまとめて扱うためのデータ構造。
- 配列を構成するそれぞれの箱を要素、何番目の箱であるかという数字を添え字またはインデックスという。配列の添え字は0から始まる。

配列の準備

- 配列を利用するためには、「配列変数の宣言」「要素の作成」という2つのステップで配列を準備しなければならない。
- 配列変数の型には「要素の型[]」を指定する。
- 要素を作成するには、「new 要素の型[要素数]」とし、配列変数に代入する。

配列の利用

- 「配列変数名[添え字]」でそれぞれの要素を読み書きできる。
- for 文や拡張 for 文を用いて配列要素に1つずつ順番にアクセスする。

配列と参照

- 配列変数は、配列の実体（new で確保された各要素のメモリ領域）を参照している。
- 特別な値 null が代入された配列変数は、どの実体も参照しない。
- 何らかの理由で参照されなくなったメモリ領域は、ガベージコレクションによって自動的に解放される。

4.9 練習問題

練習 4-1

次の条件に合った各配列を準備するプログラムを作成してください。なお、以下の 4 つの条件のコードを 1 つのプログラムの中に記述して構いません。また、値の初期化は不要です。

① int 型の値を 4 個まとめて格納できる配列 points
② double 型の値を 5 個まとめて格納できる配列 weights
③ boolean 型の値を 3 つまとめて格納できる配列 answers
④ String 型の値を 3 つまとめて格納できる配列 names

練習 4-2

次に示す 3 つの条件に沿ったプログラムを作成してください。

① 3 つの口座残高「121902」「8302」「55100」が格納されている int 型配列 moneyList を宣言します。
② その配列の要素を 1 つずつ for 文で取り出して画面に表示させます。
③ 同じ配列の要素を拡張 for 文で 1 つずつ取り出して画面に表示します。

練習 4-3

次のコードを実行すると、5 行目と 6 行目で例外が発生します。それぞれの行で発生する例外の名前を答えてください。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int[] counts = null;
4      float[] heights = { 171.3F, 175.0F };
5      System.out.println(counts[1]);
```

```
6        System.out.println(heights[2]);
7    }
8  }
```

練習4-4

次に示す4つの条件に沿って「数あてクイズ」のプログラムを作成してください。

① int型で要素数3の配列の配列numbersを準備します。このとき初期値はそれぞれ「3」「4」「9」とします。

② 画面に「1桁の数字を入力してください」と表示します

③ 次のコードを用いてキーボードから数字の入力を受け付け、変数inputに代入します。

```
int input = new java.util.Scanner(System.in).nextInt();
```

④ 配列をループで回しながら、いずれかの要素と等しいかを調べます。もし等しければ「アタリ！」と表示します。

4.10 練習問題の解答

練習 4-1 の解答

以下は解答例です（おおむね合っていれば正解として構いません）。

Main.java

```
public class Main {
  public static void main(String[] args) {
    int[] points = new int[4];
    double[] weights = new double[5];
    boolean[] answers = new boolean[3];
    String[] names = new String[3];
  }
}
```

練習 4-2 の解答

以下は解答例です（おおむね合っていれば正解として構いません）。

Main.java

```
public class Main {
  public static void main(String[] args) {
    int[] moneyList = { 121902, 8302, 55100 };
    for (int i = 0; i < moneyList.length; i++) {
      System.out.println(moneyList[i]);
    }
    for (int m : moneyList) {
      System.out.println(m);
    }
  }
}
```

練習 4-3 の解答

5 行目：NullPointerException

6 行目：ArrayIndexOutOfBoundsException

練習 4-4 の解答

以下は解答例です（おおむね合っていれば正解として構いません）。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      // (1)配列の準備
4      int[] numbers = { 3, 4, 9 };
5
6      // (2)メッセージの表示
7      System.out.println("1桁の数字を入力してください");
8
9      // (3)キーボードからの数字入力
10     int input = new java.util.Scanner(System.in).nextInt();
11
12     // (4)配列を回しながら判定
13     for (int n : numbers) {
14       if (n == input) {
15         System.out.println("アタリ！");
16       }
17     }
18   }
19 }
```

第5章

メソッド

プログラムを書いていると、コードが長くなるにつれて
全体を把握しにくくなったり、同じようなコードを
繰り返し書いたりする必要が出てきます。
そのような場合にはコードを部品化して分割すると、
全体がスッキリとして見通しのよいプログラムになります。
この章ではコードを部品化するしくみの1つである
「メソッド」について学びます。

CONTENTS

5.1 メソッドとは

5.1.1 メソッドを利用するメリット

今までの章では、mainメソッドの中にすべての文を並べてプログラムを作ってきました。これまでコードの行数は長くても30行程度でしたので、あまり問題はありませんでしたが、Javaプログラムを開発する現場では、プログラムが数千〜数万行に及ぶことも珍しくありません。

もしも、mainメソッドだけでこのような巨大なプログラムを作るとどうなるでしょうか？ たとえば開発中に「表示内容を修正してほしい」と頼まれてソースコードを修正することになったとしたら、修正箇所を探すだけでも大変な作業になることは容易に想像できますね。

Javaでは、このような不便がないように1つのプログラムを複数の部品に分けて作ることができるようになっています。本章で学ぶメソッド（method）とは、複数の文をまとめ、それを1つの処理として名前を付けたもので、部品の最小

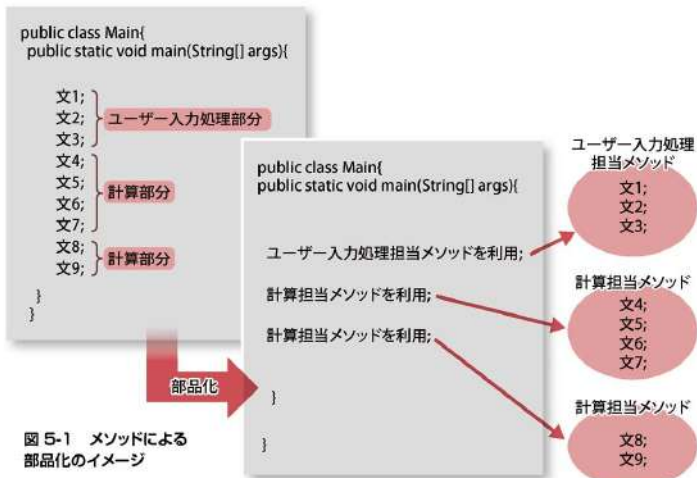



図5-1　メソッドによる部品化のイメージ

単位です。

たとえば、図5-1のようにmainメソッド内の処理を複数のメソッドに分割して処理を担当させることができます。mainメソッドは分割したメソッドを呼び出すだけになるので、コードをスッキリさせることができます。つまり機能単位でメソッドに分割することで、プログラムの「大局」を見渡せることができるようになり、全体の把握が楽になるのです。

また、メソッドに分割しておけば「表示がおかしい」など不具合が出た場合には、それを担当するメソッドを調べればよいため、修正が楽になるというメリットもあります。さらに、メソッドは繰り返し使用することができるので、同じ処理を何度も書く必要もなくなり、コードを書く手間を省くこともできます。

メソッド利用によるメリット

- プログラムの見通しがよくなり、全体を把握しやすくなる。
- 機能単位に記述するため、修正範囲を限定できる。
- 同じ処理を1つのメソッドにまとめることで、作業効率が上がる。

なんだかmainメソッドが上司で、その他のメソッドは部下みたいですね。上司が部下に仕事を振っているみたい。

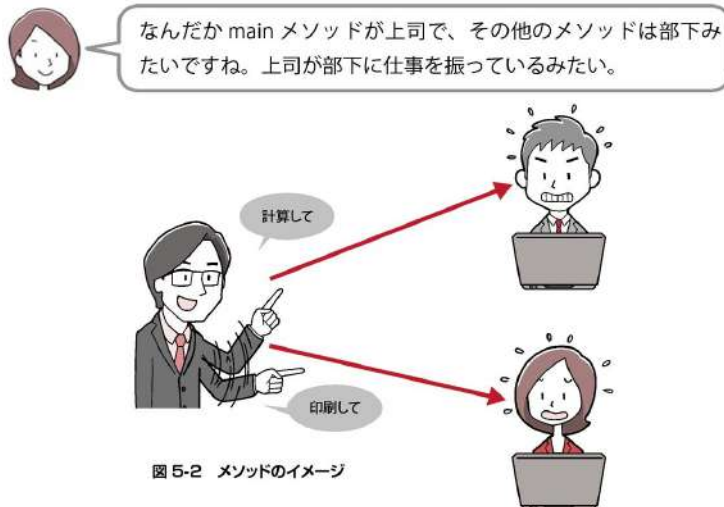



図5-2　メソッドのイメージ

5.1.2 メソッドの定義

メソッドを使用するには、まず「メソッドを作成」し、次に「作成したメソッドを使用する」という 2 つのステップが必要です。

では順に見ていきましょう。メソッドを作成することを**メソッドの定義**といい、クラスブロックで以下の構文を使用します。

メソッドの定義

public static 戻り値の型 メソッド名 (引数リスト) {
　　メソッドが呼び出されたときに実行される具体的な処理
}

public とか static、戻り値の型、引数リスト…意味のわからない単語がいっぱいありますけど…。

メソッドは文法が複雑で、ほとんどの初心者がつまずくものなんだ。だから一度に理解しようとしてはダメだよ。わからない単語は後で解説するから大丈夫。まずは一番シンプルなメソッドの定義の例から見てみよう。

リスト 5-1　シンプルなメソッドの定義

Main.java

```
1  public class Main {
2    public static void hello() {
3      System.out.println("こんにちは");
4    }
5  }
```

hello メソッドの定義（2行目）

hello メソッドが呼び出されたときの処理（3行目）

リスト 5-1 の 2 行目から 4 行目でメソッドを定義しています。「public」や「static」は決まり文句と思っておきましょう（ただし常に「public」や「static」が付くとは限りません。詳細は後の章で解説します）。また、「戻り値の型」の箇所に「void」という単語が入っていますが、これも後で説明しますので今は気にしないでください。

注目してほしいのは 2 行目の「hello」です。これは**定義するメソッドの名前（メソッド名）**になります（メソッド名の後ろの () については後で解説します）。

そして、{} の中を**メソッドブロック**と呼び、この **hello メソッドを呼び出したときに実行される**具体的な処理です。リスト 5-1 では、メソッドブロックとして画面に「こんにちは」と表示する処理を 1 行分だけ記述していますが、必要に応じて複数行の処理を記述することができます。

図 5-3　シンプルなメソッド定義の例

まずは、どこにメソッド名と処理内容を書けばよいかを覚えよう。そのほかの箇所は毎回繰り返す呪文のようなものなので、暗記しておけば OK だ。

5.1.3 メソッドの呼び出し

先ほど定義したメソッドを使ってみましょう。メソッドを使用することを**メソッドを呼び出す**といい、以下の構文を使います。

メソッドの呼び出し

メソッド名 (引数リスト)

これもメソッドの定義同様、今の段階ですべての意味を理解できなくても構いません。では、先ほどの hello メソッドを呼び出す例を見てみましょう。

リスト 5-2　メソッドの呼び出し

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("メソッドを呼び出します");
4      hello();
5      System.out.println("メソッドの呼び出しが終わりました");
6    }
7    public static void hello() {
8      System.out.println("湊さん、こんにちは");
9    }
10 }
```


Main クラスの中に main（2 行目）と hello（7 行目）の 2 つのメソッドが定義されています。このコードを実行すると、まず main メソッドが自動的に実行されます。そして、main メソッド中の 4 行目にある「hello ();」で hello メソッドが呼び出されます。これを実行した結果は次のようになります。

実行結果

```
メソッドを呼び出します
湊さん、こんにちは
メソッドの呼び出しが終わりました
```

このように、メソッドは定義しただけでは実行されず、呼び出されることで初めて、メソッドに定義した処理が実行されます。

また、図 5-4 のように、呼び出されたメソッドの処理が終了すると、メソッドの呼び出し元に戻って処理の続きが実行されていきます。

メソッドは自動的には動かないんですね。

そうだよ。利用するには必ず呼び出す必要があるんだ。呼び出し方は「メソッド名 ()」が基本だよ。

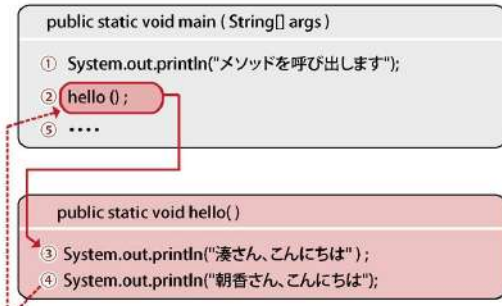



図 5-4　メソッド呼び出し時の処理の流れ

呼び出されたメソッドの中身が実行されたら、呼び出した箇所の続きが実行されるんですね。

5.1.4 main メソッド以外からのメソッドの呼び出し

メソッドは、main メソッド以外のメソッドからも呼び出すことができます。次のリスト 5-3 では methodA が methodB を呼び出しています。つまり処理の流れとしては、main、methodA、methodB の順に実行されます。

リスト 5-3　main メソッド以外からメソッドを呼び出す

Main.java

```
1  public class Main {
2    public static void methodA() {
3      System.out.println("methodA");
4      methodB();        ← methodB メソッドの呼び出し
5    }
6    public static void methodB() {
7      System.out.println("methodB");
8    }
9    public static void main(String[] args) {
10     methodA();        ← methodA メソッドの呼び出し
11   }
12 }
```

実行結果

```
methodA
methodB
```

いくつメソッドが定義されていても、main() より上に別メソッドが定義されていても、プログラムは必ず main() から動き始めるんだ。

5.2 引数の利用

5.2.1 引数とは

先輩。さっきのhelloメソッドですが、「朝香さん、こんにちは」というように別の名前を表示させる場合にはどうすればいいんですか？ 表示させたい名前の数だけメソッドを作る必要があるんですか？

それだとプログラムがメソッドだらけになってしまうね（笑）。そういった場合には「引数」を使えば解決するよ。

メソッドを呼び出す際に、呼び出し元から値を渡すことができます。このときに渡される値のことを**引数**（argument）といいます。呼び出されたメソッド側では、渡された値を受け取って処理に使用することができます。引数には数値や文字列などを指定でき、その値や型、渡す引数の数は開発者が自由に決めることができます。

図 5-5　メソッド呼び出しと同時に値を引き渡すことができる

5.2.2 1つの引数を渡す例

次のリスト5-4は、リスト5-2（p.174）のhelloメソッドに引数を渡せるように書き換えたものです。

リスト5-4　引数の例（渡す値が1つの場合）

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      System.out.println("メソッドを呼び出します");
4      hello("湊");
5      hello("朝香");
6      hello("菅原");
7      System.out.println("メソッドの呼び出しが終わりました");
8    }
9    public static void hello(String name) {
10     System.out.println(name + "さん、こんにちは");
11   }
12 }
```

"湊"を渡してhelloメソッドを呼び出す
"朝香"を渡してhelloメソッドを呼び出す
"菅原"を渡してhelloメソッドを呼び出す

リスト5-4を実行した結果は次のようになります。

実行結果

```
メソッドを呼び出します
湊さん、こんにちは
朝香さん、こんにちは
菅原さん、こんにちは
メソッドの呼び出しが終わりました
```

まずは4行目の「hello("湊");」に注目してください。()の中に「湊」という文字列の値が入っています。このようにメソッドを呼び出す際、()の中に値を入れておくと、その値が引数として呼び出される側のhelloメソッドに渡されます（同じく5行目と6行目でも、それぞれ引数として「朝香」と「菅原」がhelloメソッドに渡されます）。

次に9行目のhelloメソッドの定義に注目してください。メソッド名の後ろにある()の中で、「String name」として文字列変数nameを宣言しています。helloメソッドが呼び出されると、このnameに呼び出し元から引数として渡された値「湊」が自動的に代入され、メソッドブロック内で使用することができるようにな

ります。この例では画面への出力に使用されており、その結果「湊さん、こんにちは」と表示されます。

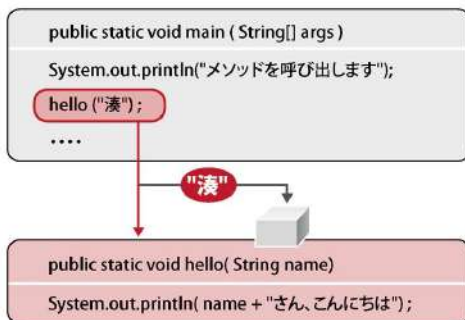



図 5-6　引数が 1 つの場合のイメージ

リスト 5-2 の hello メソッドの場合は () の中に何も書いてないから値は渡していないんですね。

そのとおりだ。何も渡さないときでも () は絶対に書く必要があるから気をつけよう。() は「何も渡さない」ということを意味しているよ。

これで表示する名前を自由に変えることができるようになりました。引数が使えるようになると便利ですね！！

渡す値は 1 つだけでなく複数の値を渡すこともできるんだ。それができると、もっと便利になるよ。

5.2.3 複数の引数を渡す例

次は引数が 2 つになった場合を見てみましょう。

リスト 5-5　引数の例（渡す値が複数の場合）

Main.java

```
public class Main {
  public static void main(String[] args) {
    add(100, 20);
    add(200, 50);
  }
  // 複数の値を受け取るaddメソッド
  public static void add(int x, int y) {
    int ans = x + y;
    System.out.println(x + "+" + y + "=" + ans);
  }
}
```

3行目：100 と 20 を渡して add メソッドを呼び出す
4行目：200 と 50 を渡して add メソッドを呼び出す

リスト 5-5 を実行した際のイメージは図 5-7 のようになります。

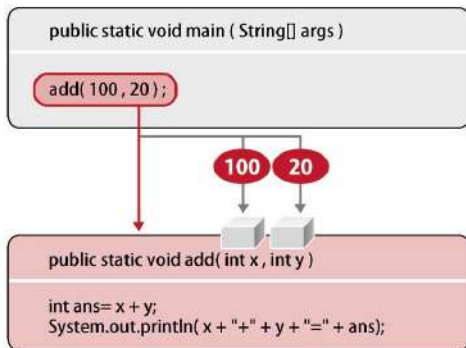



図 5-7　引数が 2 つの場合のイメージ

引数として渡す値が複数ある場合、リスト 5-5 の 3、4 行目のように、**値をカンマで区切って使用します**。また、値を受け取るメソッド側でも、受け取る変数をカンマで区切って宣言します（リスト 5-5 の 7 行目）。このとき、引数として渡される値と、メソッド側で宣言する変数の型と順番を合わせておく必要があることに注意してください。たとえば図 5-8 にあるように、文字列型の値を整数型

変数で受け取るなど、引数と変数の型が合致しない場合にはコンパイルエラーが発生します。

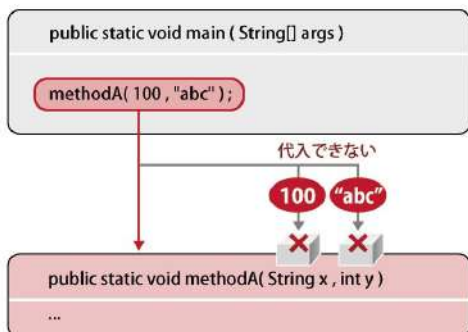



図 5-8　引数と受け取る側の変数で型が異なる場合には代入できずエラーになる

渡すほうも受け取るほうも、カンマで区切って指定するんですね。

それに、メソッドの定義を踏まえて渡す引数を指定しないとダメなのね。

引数の渡し方

何も渡さない場合:メソッド名 ()

値を１つ渡す場合:メソッド名 (値)

値を複数渡す場合:メソッド名 (値 , 値 , …)

※値には、変数名を指定することもできる。

5.2.4 仮引数と実引数

ここで、再度メソッド定義と呼び出しの構文を見直してみましょう。

メソッドの定義（再掲）

public static 戻り値の型 メソッド名 (引数リスト) {
　　メソッドが呼び出されたときに実行される具体的な処理
}

※引数リストには、呼び出し元から渡された値を入れる変数宣言をカンマ区切りで並べる。

メソッドの呼び出し（再掲）

メソッド名 (引数リスト)

※引数リストには、呼び出し時に渡したい具体的な値や変数をカンマ区切りで指定する。

値を渡すのも、受け取るのも、共に () 内の「引数リスト」の箇所で行います。**渡す値、受け取る変数ともに「引数」**と呼ばれますが、細かく呼び分ける場合、渡す値のことを**実引数**、受け取る変数のことを**仮引数**と呼びます。

リスト 5-5（p.180）では 3 行目の 100 と 20、および 4 行目の 200 と 50 が実引数です。そして 7 行目の x と y が仮引数になります。

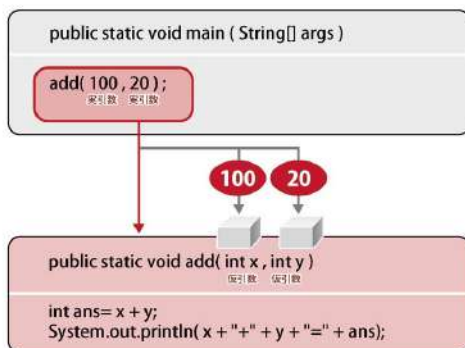



図 5-9 実引数と仮引数

5.2.5 変数のスコープとローカル変数

うーん。引数って便利だけど、ルールが多くてややこしいですね。そんなことしなくても main メソッドで用意した変数を使えばいいんじゃないんですか？

湊くんが言っているのは、次のリスト 5-6 のようなことです。

リスト 5-6 引数を使わずにできないのか？

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int x = 100;          ← add メソッドで使用するつもり
4      int y = 10;           ← add メソッドで使用するつもり
5      add();                ← add メソッドを呼び出す
6    }
7    public static void add() {
8      int ans = x + y;      ← ここで使用するつもり（エラー）
```

```
 9          System.out.println(x + "+" + y + "=" + ans);
10      }
11  }
```

確かに引数はややこしいね。でも、このコードだとコンパイルエラーになるよ。

あっ！ 本当だ。コンパイルしたら add メソッド内で「x と y が見つけられません」って怒られました。

これは変数のスコープが原因なんだよ。

スコープって前に教わったような…。

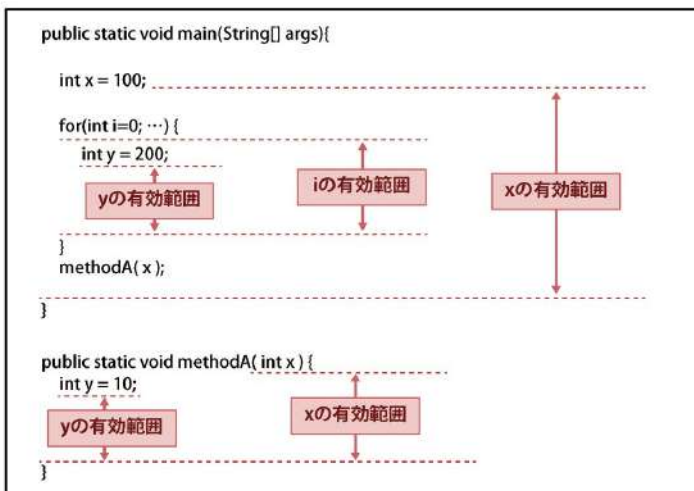
```
public static void main(String[] args){

   int x = 100;

   for(int i=0; …) {
      int y = 200;
   }
   methodA( x );
}

public static void methodA( int x ) {
   int y = 10;
}
```


図 5-10　変数のスコープ（有効範囲）

第３章で変数のスコープ（有効範囲）について学びました（忘れた人は 3.2.1 項を読み返してください）。ブロック内で宣言された変数のスコープは、そのブロック内に限定されるのがルールでしたね。そのため **main メソッド内で宣言した変数 x と y は、main メソッドのブロックの中でしか使用できません**。これを add メソッドで使おうとすると、x と y を見つけることができずエラーになります。

なるほど。だから main メソッドで宣言した x と y を add メソッドで使おうとしたらエラーになったわけですね。

なお、main や add といったメソッド内で宣言した変数を**ローカル変数**と呼び、仮引数もその一種です。

ローカル変数は、その変数が属するメソッド内だけで有効な存在であって、**別のメソッドに属する同名のローカル変数とはまったくの別物**です。たとえば、図 5-10 のコードにある「main メソッド内の変数 y」と「methodA 内の変数 y」の２つは、名前は同じですが無関係です。methodA の変数 y にどんな値を代入しても、main メソッドの変数 y にはいっさい影響はありません。

ローカル変数の独立性

異なるメソッドに属するローカル変数は、お互いに独立していて無関係

5.3 戻り値の利用

5.3.1 戻り値とは

呼び出されたメソッドから、呼び出し元のメソッドへ値を返すことを**値を戻す**（または値を返す）といいます。また、戻されるデータ（値）のことを**戻り値**（または「返り値」）といいます。

図 5-11　戻り値のイメージ

値を戻す場合、以下の構文を使用します。

値の戻し方

```
public static 戻り値の型 メソッド名 ( 引数リスト… ) {
    メソッドが実行されたときに動く処理
    return 戻り値 ;
}
```

まずはメソッド内の「return」の行に注目してください（**return 文**といいます）。これにより、return の後ろに書かれた値を呼び出し元に戻すことができます。変数に入っている値を戻すには、「return 変数名 ;」のように変数名を指定します。また「return 100;」や「return "hello";」のように戻したい値を直接書いても構いません。

次にメソッド名の左側にある「**戻り値の型**」に注目してください。ここには、基本的に return によって戻される値と同じ型を指定します。100 のような整数を戻す場合は「int」、"hello" のような文字列型を戻す場合は「String」、変数に入っている値を戻す場合は、その変数の型を書きます。**何も戻さない場合は「void」を指定します**。void は「何もない」という意味です。

ということは、たとえば ans に入っている計算結果を戻したい場合は、「 return ans; 」をメソッド内に書き足して「戻り値の型」を「void」から「int」にすればいいのね。

今まで見てきた例が「void」だったのは、何も戻していなかったからなんだね。

5.3.2 戻り値を受け取る

メソッド側の書き方はわかったけど、どうやって呼び出し側で受け取ればいいんだろう？

戻り値は引数と同様に、変数を用意して受け取る必要があります。受け取るには、呼び出し元で以下の構文を使用します。

メソッドを呼び出し、戻り値を受け取る

型 変数名 = メソッド名 (引数リスト) ;

文に代入演算子（p.71 の 2.4.3 項）「=」がある場合、常に右辺から先に評価されるので、まずはメソッドの呼び出しが実行されます。呼び出されたメソッドが return 文によって値を戻す場合、**「メソッド名（引数リスト）」という部分は評価さ**

れて戻ってきた値に置き換わります。それによって「型 変数名 = メソッドの戻り値;」という状態になり、戻り値が変数に代入されます。

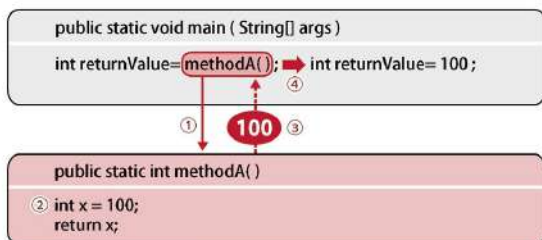



図 5-12　戻り値の受け取り

戻り値を受け取る変数は 1 つしか用意できないんですか？

そうだよ。引数みたいに複数用意できないから注意してね。

はい、さっそく戻り値を使うように add メソッドを作り直してみました（リスト 5-7）。

リスト 5-7　戻り値の例

Main.java

```
1  public class Main {
2    public static int add(int x, int y) {
3      int ans = x + y;
4      return ans;
5    }
6    public static void main(String[] args) {
7      int ans = add(100, 10);
8      System.out.println("100 + 10 = " + ans);
9    }
10 }
```

add メソッドの呼び出し（110 に化ける）

リスト 5-5 の add() は計算のあとに表示までしてしまうメソッドだったけど、リスト 5-7 の add() は計算だけをして結果を呼び出し元に返しているね。

main メソッド側ではこの戻り値を他の計算や表示に自由に利用できますね。

5.3.3 戻り値をそのまま使う

メソッドの戻り値を変数で受けずに、そのまま使うこともできます。ちょっと極端な例ですが、次のリスト 5-8 を見てください。

リスト 5-8 戻り値をそのまま使う

Main.java

```
1 public class Main {
2   public static int add(int x, int y) {
3     int ans = x + y;
4     return ans;
5   }
6   public static void main(String[] args) {
7     System.out.println(add(add(10, 20), add(30, 40)));
8   }
9 }
```

30 に化ける　70 に化ける

リスト 5-8 の 7 行目には add メソッドの呼び出しが 3 つあります。順に見ていきましょう。まず「 add (10 , 20) 」と「 add (30 , 40) 」が実行され、それぞれの計算結果である「30」と「70」が戻り値として呼び出し元に返されます。これによりカッコの外側の add メソッドは「add (30 , 70) 」という状態になります。そして、「30」と「70」の引数を持って add メソッドが再び呼び出され、「100」が add メソッドより戻されます。最終的に「 System.out.println (100); 」という状態になり、画面には「100」が出力されます。

呼び出したメソッドに戻り値があっても、絶対に変数で受け取らないといけないというわけではないんですね。

そうだよ。場合によっては、戻り値があるけど使わないときもあるんだ。そのときは、「メソッド名(引数リスト)」で呼び出すだけでいいよ。

5.3.4 return 文の注意点

return 文は値を戻すだけでなく、**メソッドの終了も行います**。そのため、return 文の後に処理を書いても実行ができません(コンパイルエラーになります)。うっかりリスト 5-9 のようなコードを書かないようにしましょう。

リスト 5-9　誤った return 文

```
1 public static int sample() {
2   ⋮
3    return 1;
4    int x = 10;
5 }
```

この文が実行されることはないのでコンパイルエラーになる

5.4 オーバーロードの利用

5.4.1 類似する複数のメソッドを定義する

開発をしていると「似たような処理を行うメソッドを複数作る」必要に迫られることがあります。しかし、処理内容が似ているからといって、**メソッドに同じ名前を付けることは基本的にできません。**同じ名前のメソッドが複数あると、JVMはどれを実行してよいか判断できないのでコンパイルエラーになります（図5-13）。

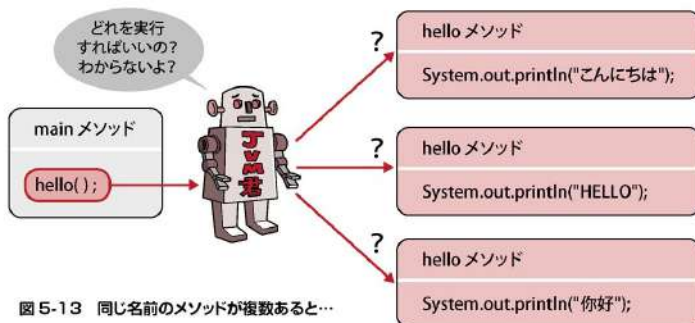



図5-13　同じ名前のメソッドが複数あると…

でも、処理内容が似ているのに名前を同じにできないなんて不便ですね。処理内容が似ていると同じ名前を付けたくなります。

そうだね。だから、例外的に同じ名前のメソッドを複数定義する方法があるんだ。

同じ名前のメソッドを定義することを**オーバーロード**（overload）（または**多重定義**）といいます。次ページのリスト5-10を見てください。

リスト 5-10　オーバーロード（引数の型が異なる場合）

Main.java

```
public class Main {
  // 1つ目のaddメソッド
  public static int add(int x, int y) {
    return x + y;
  }
  // 2つ目のaddメソッド
  public static double add(double x, double y) {
    return x + y;
  }
  // 3つ目のaddメソッド
  public static String add(String x, String y) {
    return x + y;
  }
  public static void main(String[] args) {
    System.out.println(add(10, 20));
    System.out.println(add(3.5, 2.7));
    System.out.println(add("Hello", "World"));
  }

}
```

1つ目のaddメソッドが呼び出される

2つ目のaddメソッドが呼び出される

3つ目のaddメソッドが呼び出される

実行結果

```
30
6.2
HelloWorld
```

addメソッドが3つ定義されていることに注目してください。それぞれの仮引

数の型を見ると、1つ目は「int , int」、2つ目は「double , double」、3つ目は「String , String」と、それぞれ異なっています。

このように、**仮引数が異なれば同じ名前のメソッドを複数定義することが許されています。**同じ名前のメソッドが複数あったとしても、仮引数の型が異なっていれば、JVMが呼び出し元の引数(実引数)を見て、その引数の型に一致するメソッドを呼び出してくれるのです(図5-14)。

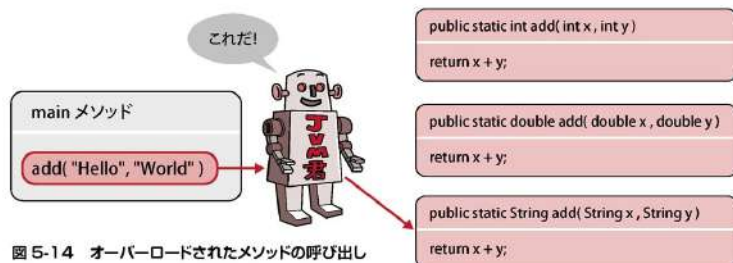



図5-14 オーバーロードされたメソッドの呼び出し

また、仮引数の型だけでなく、個数が違う場合もオーバーロードできます。次のリスト5-11にはaddメソッドが2つありますね。1つ目はint型の仮引数が2つ、2つ目はint型の仮引数が3つあります。addメソッドが呼び出される際に、JVMは引数の型と個数を比較して一致するほうのaddメソッドを呼び出してくれます。

リスト5-11 オーバーロード(引数の数が異なる場合)

Main.java

```
1  public class Main {
2    public static int add(int x, int y) {        ← 1つ目のaddメソッド
3      return x + y;
4    }
5    public static int add(int x, int y, int z) { ← 2つ目のaddメソッド
6      return x + y + z;
7    }
8    public static void main(String[] args) {
9      System.out.println("10+20=" + add(10, 20)); ← 1つ目のaddメソッドが呼び出される
```

```
10          System.out.println("10+20+30=" + add(10, 20, 30));
11      }
12  }
```

2つ目の add メソッドが呼び出される

次の実行結果を見ると、2つの add メソッドが区別され、正しく呼び出されていることがわかります。

実行結果

```
10+20=30
10+20+30=60
```

オーバーロード

仮引数の個数か型が異なれば、同じ名前のメソッドを複数定義できる（引数は同じで、戻り値の型だけが異なる場合は不可）。

メソッドのシグネチャ

メソッド宣言で「戻り値の型」の後に記述する以下の情報をまとめて、メソッドの**シグネチャ** (signature) といいます。

①メソッド名　　②引数の個数・型・並び順

オーバーロードは「シグネチャが重複しない場合のみ許される」と覚えておいてもよいでしょう。

5.5 引数や戻り値に配列を用いる

5.5.1 引数に配列を用いる

メソッドの引数には int 型や String 型などの変数だけでなく、リスト 5-12 のように配列も使うことができます。

リスト 5-12 配列が引数

Main.java

```
public class Main {
  // int型配列を受け取り、すべての要素を表示するメソッド
  public static void printArray(int[] array) {
    for (int element : array) {
      System.out.println(element);
    }
  }
  public static void main(String[] args) {
    int[] array = { 1, 2, 3 };
    printArray(array);     // 配列を渡す
  }
}
```

int 型の変数を受け取りたい場合は int と書いていたけど、配列の場合は int[] にすればいいんだ！

5.5.2 値渡しと参照渡し

そういえば、int[] 型のような配列型変数には、
配列の実体を指し示すメモリ番地が入っているんでしたよね？

ということは、リスト 5-12 で引数として渡しているのは
「配列まるごと」ではなく、アドレス情報だけなんですか？

そのとおり。引数として値ではなくアドレスを渡すと、
ちょっと不思議な現象が起こるので、ここで紹介しておこう。

メソッドを呼び出すときに引数として変数を指定した場合、メソッドに渡されるのは変数そのものではなく、変数に入っている値です（正確には、メソッドを呼び出した時点での変数の値が、メソッドの仮引数にコピーされます）。このように、**値そのものが渡される呼び出し**を**値渡し**（call by value）と呼びます。

たとえば次の図 5-15 の場合、methodA が呼び出される際に main メソッドの変数 x の内容（=100）が引数 x にコピーされるため、2 つの変数の中身は同じ 100 になります。しかし、この 2 つの x は「まったく別の存在」ですので、methodA の中で引数 x の中身にどんな値を代入しようとも、呼び出し元である main メソッド内の変数 x の中身は 100 のまま変わることはありません。

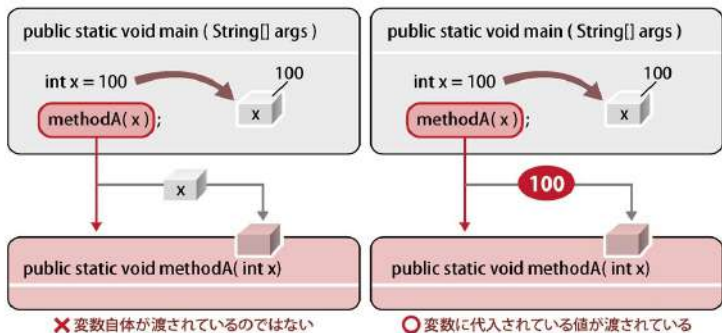



図 5-15　値渡しの場合、値がコピーされる

基本データ型の変数をメソッド呼び出しで渡すと

- 呼び出し元の変数の内容が、呼び出し先の引数にコピーされる。
- 呼び出し先で引数の内容を書き換えても、呼び出し元の変数は変化しない。

しかし、メソッド呼び出しの際に普通の変数ではなく配列を渡すと不思議なことが起こります。その原理を次ページの図 5-16 で順に見ていきましょう。

まず、メソッド呼び出しの際にコピーされるのは、配列の内容(=1,2,3)ではなく配列の先頭要素のアドレス(=8832 番地)です。すると、main メソッド内の変数 array と printArray メソッド内の引数 array はどちらも 8832 番地以降にある配列の実体を参照した状態になります。

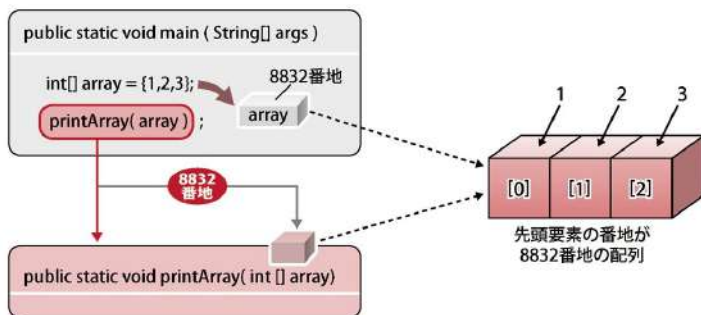



図 5-16　参照渡しの場合、配列の先頭要素のアドレスが渡される

1 つの配列を複数の配列変数で参照することになるんですね。これって、配列のときに学んだ「配列を複数の変数で参照」するしくみ(4.5.4 項)と同じじゃないですか？

そうだよ。よく気づいたね。

この状態で、printArray メソッド内で array[0] に 100 を代入したらどうなるでしょうか？　もちろん、8832 番地にある要素が 100 に書き換わりますね。では printArray メソッドが終了した後、main メソッド内で array[0] を取り出したらどうなるでしょうか？　8832 番地にある要素の値、つまり 100 を取り出すことになるのです。

今回の配列の例のように、引数としてアドレスを渡すことを**参照渡し**（call by reference）といいますが、参照渡しを行うと「呼び出し先で加えた変更が呼び出し元にも影響する」ようになります（なお、Java のこのしくみは、厳密には「参照の値渡し」といわれるもので、狭義の参照渡しと区別することがあります）。

配列をメソッド呼び出しで渡すと

- 呼び出し元の配列のアドレスが、呼び出し先の引数にコピーされる。
- 呼び出し先で配列の実体を書き換えると、呼び出し元にも影響する。

参照渡しによって発生するこの不思議な現象を体験するために、ぜひリスト 5-13 を実行し、「2」「3」「4」と表示されることを確認してみてください。

リスト 5-13　同じ配列を参照していることを確認する

Main.java

```
1  public class Main {
2    // int型配列を受け取り、
3    // 配列内の要素すべてに1を加えるメソッド
4    public static void incArray(int[] array) {
5      for (int i = 0; i < array.length; i++) {
6        array[i]++;
7      }
8    }
9    public static void main(String[] args) {
10     int[] array = { 1, 2, 3 };
11     incArray(array);
```

計算結果を return で返していない

メソッド実行

```
12        for (int i : array) {
13          System.out.println(i);
14        }
15    }
16  }
```

array の全要素を出力

引数に普通の変数を渡すときと違って、呼び出し先のメソッドでの変更が呼び出し元に影響を与えるんですね。

これは、配列型変数などの参照型変数の特徴だよ。よく覚えておこう。

5.5.3 戻り値に配列を用いる

引数と同様に、戻り値に配列を使用することができます（リスト 5-14）。

リスト 5-14　戻り値が配列の場合

Main.java

```
1   public class Main {
2     public static int[] makeArray(int size) {
3
4       int[] newArray = new int[size];
5       for (int i = 0; i < newArray.length; i++) {
6         newArray[i] = i;
7       }
8       return newArray;
9     }
10    public static void main(String[] args) {
11      int[] array = makeArray(3);
12      for (int i : array) {
13        System.out.println(i);
14      }
```

int 型配列を作成して戻すメソッド

配列を戻す

array の全要素を出力

```
15      }
16  }
```

実行結果

```
0
1
2
```

int 型を戻す場合は戻り値の型は「int」、int 型配列を戻す場合は int[] ですね。

そうだ。これも引数と同様で、実際には配列そのものを戻しているわけではなくて、図 5-17 のように配列のアドレスを戻しているんだ。

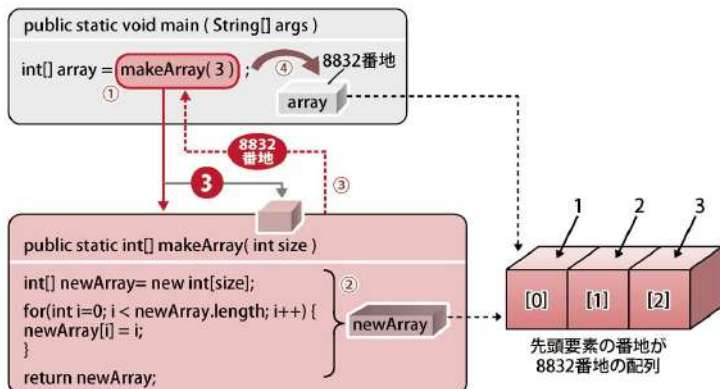



図 5-17　戻り値が配列型変数の場合、配列の先頭要素のアドレスが戻される

リスト 5-14 の 8 行目「return newArray;」によって配列の先頭要素のアドレスが main メソッドに戻されます。main メソッドでは、それを自身で宣言した配列変数 array に代入します。その結果、makeArray メソッドで作成された配列を参照できるようになります。

5.6 コマンドライン引数

そういえば、main メソッドも String 型の配列を引数としているじゃない。これには何が入るのかな？

そんなの「main メソッドを呼び出すメソッドが指定した実引数」に決まっているじゃないか。

でもほら、最初に動く main メソッドって、どのメソッドからも呼ばれないでしょう？

5.6.1 コマンドライン引数

朝香さんが気づいたように、main メソッドは文字列配列を引数として受け取るように定義されています。

```
public static void main(String[] args) {
```

通常のメソッドの場合、仮引数に入ってくる値は「呼び出し元のメソッド」が指定した実引数です。しかし最初に動く main メソッドには「呼び出し元のメソッド」がありません。main メソッドには、いったい「誰」が、「どんな」引数を渡すことができるのでしょうか。

実は、Java のプログラムを起動する際、さまざまな「追加情報」を指定して起動することができます。このプログラム起動時の追加情報のことを**コマンドライン引数**（command line argument）といいます。「dokojava」を利用している方は実行画面でコマンドライン引数を指定することができますし、この後の付録 A で紹介する、java コマンドを使ってプログラムを実行する方法では、次のようにプログラム名の後ろにコマンドライン引数を指定できます。

java　プログラム名　引数リスト（半角スペース区切り）

たとえば、「java Main ミナト 勇者」のように主人公の名前と職業を指定して起動できるようなRPGプログラムなども作れるのです。

そして、いざプログラムが起動すると、JVMは半角で区切られた情報の1つ1つを配列に詰め込んで実引数とし、mainメソッドを起動してくれます（図5-18）。

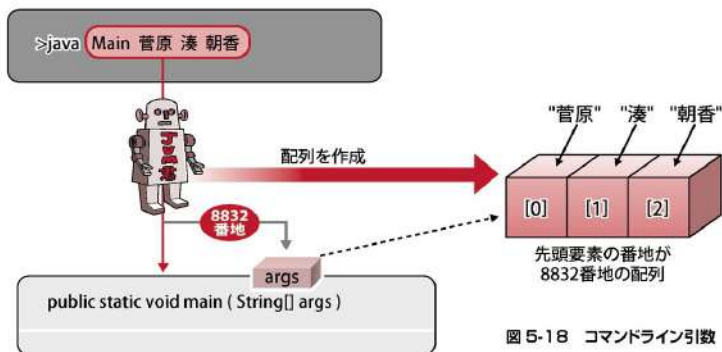



図5-18　コマンドライン引数

図5-18では、起動時のコマンドライン引数として「菅原 湊 朝香」を指定しています。よって、args [0] には「菅原」が、args [1] には「湊」が、そしてargs [2] には「朝香」が格納され、args.length は3になります。

コマンドライン引数

プログラム起動時に指定したコマンドライン引数が、JVMによって配列に変換され、mainメソッド起動時に渡される。

5.7 第５章のまとめ

この章では、次のようなことを学びました。

メソッド

- メソッドでコードを部品化することができる。
- クラスブロックの中にメソッド定義を宣言することができる。

引数

- メソッドの呼び出し時に、引数として値を渡すことができる。
- メソッドを呼び出すときに渡す値を実引数、受け取る側の変数を仮引数という。
- メソッド内で宣言した変数はローカル変数といい、ほかのメソッドからは使用できない。また、そのメソッドの実行が終わるとローカル変数は消滅する。

戻り値

- return 文を使用してメソッドの呼び出し元へ値を戻すことができる。
- 戻す値の型はメソッドの定義で宣言する必要がある。
- 戻り値を受け取るには代入演算子「=」を使用する。

メソッドの活用

- 仮引数の数と型が異なる同じ名前のメソッドを定義することができる（オーバーロード）。
- 配列を渡すとき、あるいは戻すときは、配列そのものではなく配列のアドレスを渡している。

5.8 練習問題

練習 5-1

以下の仕様を参考にメソッド「introduceOneself」を定義してください。

メソッド名	introduceOneself
戻り値の型	なし
引数リスト	なし
処理内容	名前（文字列）、年齢（整数）、身長（浮動小数点）、性別（1 文字）を代入する変数を宣言して値を代入する。変数を利用して自己紹介を表示する。表示するデータの中身や表示の仕方は自由とする。

練習 5-2

以下の仕様を参考にメソッド「email」を定義してください。

メソッド名	email
戻り値の型	なし
引数リスト	メールのタイトル（String title） メールの宛先アドレス（String address） メールの本文（String text）
処理内容	以下の形式で表示を行う（色文字の箇所は引数を使用すること）。 「メールの宛先アドレス」に、以下のメールを送信しました 件名：「メールのタイトル」 本文：「メールの本文」

練習 5-3

次の仕様を参考にして、練習 5-2 のコードにメソッド「email」をオーバーロードし、main メソッドから呼び出してください。

メソッド名	email
戻り値の型	なし
引数リスト	メールの宛先アドレス (String address) メールの本文 (String text)
処理内容	以下の形式で表示を行う（色文字の箇所は引数を使用すること)。 「メールの宛先アドレス」に、以下のメールを送信しました 件名:無題 本文:「メールの本文」

練習 5-4

以下の仕様を参考にメソッド「calcTriangleArea」と「calcCircleArea」を作成してください。

メソッド名	calcTriangleArea
戻り値の型	三角形の面積（double）
引数リスト	三角形の底辺の長さ、単位は cm（double bottom） 三角形の高さ、単位は cm（double height）
処理内容	引数を使用して面積を求め、それを戻す。

メソッド名	calcCircleArea
戻り値の型	円の面積（double）
引数リスト	円の半径、単位は cm（double radius）
処理内容	引数を使用して面積を求め、それを戻す。

main メソッドからそれぞれのメソッドに適当な引数を渡して呼び出し、戻り値を出力して正しい面積が表示されるかを確認してください。

（例）

- **三角形の底辺の長さが 10.0cm、高さが 5.0cm の場合、面積は 25.0cm^2**
- **円の半径が 5.0cm の場合、面積は 78.5cm^2**

練習問題の解答

練習 5-1 の解答

以下は解答例です。おおむね合っていれば正解とします。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      introduceOneself();
4    }
5    public static void introduceOneself() {
6      String name = "Java";
7      int age = 34;
8      double height = 169.9;
9      char gender = '男';
10     System.out.println("私の名前は" + name + "です");
11     System.out.println("歳は" + age + "歳です");
12     System.out.println("身長は" + height + "cmです");
13     System.out.println("性別は" + gender + "です");
14   }
15 }
```

練習 5-2 の解答

以下は解答例です。おおむね合っていれば正解とします。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String title = "お誘い";
4      String address = "uso800@xxxx.com";
```

```
 5      String text = "今度、飲みにいきませんか";
 6      email(title, address, text);
 7    }
 8    public static void email(String title, String address,
          String text ) {
 9      System.out.println
            (address + " に、以下のメールを送信しました");
10      System.out.println("件名：" + title);
11      System.out.println("本文：" + text);
12    }
13  }
```


練習 5-3 の解答

以下は解答例です。おおむね合っていれば正解とします。

Main.java

```
 1  public class Main {
 2    public static void main(String[] args) {
 3      String address = "uso800@xxxx.com";
 4      String text = "今度、飲みにいきませんか";
 5      email(address, text);
 6    }
 7    public static void email(String address, String text) {
 8      System.out.println
            (address + "に、以下のメールを送信しました");
 9      System.out.println("件名:無題");
10      System.out.println("本文:" + text);
11    }
12    public static void email(String title, String address,
          String text) {
13      System.out.println
            (address + "に、以下のメールを送信しました");
```

```
14      System.out.println("件名：" + title);
15      System.out.println("本文：" + text);
16    }
17  }
```

練習 5-4 の解答

以下は解答例です。おおむね合っていれば正解とします。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      double triangleArea = calcTriangleArea(10.0, 5.0);
4      System.out.println
           ("三角形の面積：" + triangleArea + "平方cm");
5      double circleArea = calcCircleArea(5.0);
6      System.out.println("円の面積：" + circleArea + "平方cm");
7    }
8    public static double calcTriangleArea(double bottom,
         double height ) {
9      double area = (bottom * height) / 2;
10     return area;
11   }
12   public static double calcCircleArea(double radius) {
13     double area = radius * radius * 3.14;
14     return area;
15   }
16 }
```

付録 A

JDK を用いた開発

第 5 章まで Java プログラミングの基礎について解説してきました。
そして次の第 6 章からは本格的な開発へと進んでいきます。
しかしその前に、必要なツールを準備しておきましょう。
これまでみなさんが使ってきた「dokojava」は、入門学習用に適した
便利な開発ツールではありますが、あくまで簡易的なものです。
本格的な Java プログラミングでは JDK と呼ばれるツールを使います。
本章では、そのインストールと開発に必要な準備について解説します。

CONTENTS

A.1 Javaの開発に必要なツール

A.1.1 JDKのインストール

第1章で解説したように、Javaのプログラム開発は以下の3つのステップで進みます。

①ソースコードの作成と編集
②コンパイルによる変換
③完成プログラムの実行

ここまで、私たちは上記①〜③のすべてをdokojavaで実行してきました。しかし、より本格的なJavaプログラムを開発していくにあたって、dokojavaはやや力不足です。ここからは、本格的な開発環境をPCにインストールしてそれを使っていきましょう（第6章以外はdokojavaでも学習できますが、ここで開発環境をインストールして、第6章以降でも使っていくことをお薦めします）。

PCで開発を行う場合、手順①には「**テキストエディタ**」、手順②には「**Javaコンパイラ**」、手順③には「**Javaインタプリタ**」と呼ばれる開発用ツールを利用します。

各開発手順で必要となるツール

手順	使うツールの一般名称	代表製品の名称
①ソース作成	テキストエディタ	Windows:メモ帳 Linux:vi、emacs Mac:テキストエディット
②コンパイル	Javaコンパイラ	javac
③実行	Javaインタプリタ	java

使用するテキストエディタは使い慣れたもので構いません。代表的なものとして、Windowsでは「メモ帳」、LinuxやMacではviやemacs、テキストエディットがあります。

テキストエディタの準備ができたら、次はJavaの開発環境をインストールし

ます。Java のインタプリタやコンパイラ、そのほか関連の開発ツール一式を詰め合わせたパッケージ「JDK」(Java SE Development Kit)はオラクル社の Web サイトから無料でダウンロードできます(ダウンロードする Web サイトは後述する A.1.3 項を参照してください)。これをダウンロードしてセットアップすれば、自分の PC で開発を行うために必要な道具が揃います。

A.1.2 JDK セットアップの複雑さ

JDK のセットアップは、パソコン初心者にとっては少し複雑な作業です。「JDK のセットアップ方法がわかりません」「セットアップしてみたのですがうまく動きません」という質問を、とてもよく耳にします。次のような事情から、初心者にとって JDK のセットアップはやや難しいものとなっています。

- **JDK をダウンロードする Web サイトの URL は変更されることがある。**
- **JDK の導入後に、やや高度な OS 設定(環境変数の変更)が必要となる。**
- **パソコンの種類によって、設定する項目の名称が異なる。**

A.1.3 「JDK セットアップガイド」の利用

本書では、初心者であっても前述の 3 つの課題を乗り越えて JDK を無事セットアップできるように、「JDK セットアップガイド」を用意しています。Web ブラウザで次のアドレスにアクセスすると、セットアップ手順が表示されますので、ぜひ利用してください。

JDK セットアップガイド

http://dokojava.jp/jdk

ガイドにアクセスすると、OS の種類に従って、該当するページが表示されます。ガイドの指示に従いながら 1 ステップずつ進んでいけば、JDK のダウンロードとインストール、環境変数の設定、そして動作確認まで行えるようになっています。

A.2 コマンドラインプロンプトの使い方

A.2.1 コマンドラインプロンプトとは

JDK で Java プログラムを開発するには「コマンドプロンプト」に関する知識が必要だ。ここで簡単に紹介しておこう。

コマンドラインプロンプトとは、コンピュータに文字で指示を行うためのプログラムです。Windows では「コマンドプロンプト」、Mac や Linux では「ターミナル」「端末」などと呼ばれます。Java の開発は基本的に、このコマンドプロンプト画面で行います。

図A-1　Windows7のコマンドプロンプト画面

コマンドプロンプトを使うことで、プログラムのコンパイルと実行、その他ファイルのコピーなど、さまざまな指示をコンピュータに送ることができます。

A.2.2 コマンドプロンプトの起動

Windows でコマンドプロンプトを起動する場合、画面左下の「スタート」メニューから「すべてのプログラム」→「アクセサリ」→「コマンドプロンプト」を選択します。

Mac の場合は「アプリケーション」→「ユーティリティ」→「ターミナル」で起動します。Linux の場合はディストリビューションごとに異なりますので、マニュ

アルを参照してください。

さて、コマンドプロンプトを起動すると、次のような文字が表示されます（表示内容は環境によって異なります）。

```
C:¥Users¥sugawara>
```

この「>」記号で終わる表示をプロンプトと言い、この記号の右側にコマンドを入力してコンピュータに対して指示を送ります。

A.2.3 カレントディレクトリ

プロンプトに表示されている「C:¥Users¥sugawara」は、**カレントディレクトリ**（current directory）と呼び、「現在着目しているフォルダ」を示しています。

先ほどの場合は「Cドライブの、Usersフォルダの中の、sugawaraフォルダ」に着目していることを意味します。Javaのプログラムを作成・実行するには、この着目しているフォルダの中のファイルを編集・実行します。また、必要に応じて別のフォルダにも着目しながら開発作業を進めていきます。

A.2.4 ファイルの一覧表示

コマンドプロンプトを操作してフォルダの中を見てみましょう。Windowsでは「dir」、MacとLinuxでは「ls -la」とコマンドを入力します（↵マークのところは、Enterキーを押します）。

```
（※これはWindowsの画面です）
C:¥Users¥sugawara>dir↵
2011/08/15 10:52 <DIR>          Contacts
2011/08/15 10:52 <DIR>          Desktop
2011/08/15 10:52 <DIR>          Documents
        ⋮
2011/08/15 10:52 <DIR>          Links
2011/08/15 10:54             25 Main.java
C:¥Users¥sugawara>
```

カレントディレクトリに含まれている、すべてのファイルとフォルダの一覧が表示されます。なお、<DIR>と書かれている行はフォルダ、そうではない行はファイルを表しています。今回の場合、DesktopやLinksのようなフォルダと、Main.javaというファイルがあることがわかります。

A.2.5 カレントディレクトリの変更

着目するフォルダを変更するには、cdコマンドを使います。たとえば、現在のフォルダの中にあるDesktopフォルダに移動するには、以下のように入力します。

```
C:¥Users¥sugawara>cd Desktop ↵
C:¥Users¥sugawara¥Desktop>
```

カレントディレクトリが変わった

また、現在着目しているフォルダの1つ上のフォルダに移動するには、次のように「..」を指定します。

```
C:¥Users¥sugawara¥Desktop>cd .. ↵
C:¥Users¥sugawara>
```

1つ上のフォルダに移動した

A.2.6 その他の操作

コマンドプロンプトでは、その他にも以下のようなコマンドを使ってコンピュータに指示を与えることができます。

【ファイルのコピー】	copy	コピー元ファイル名	コピー先ファイル名
【ファイル名の変更】	ren	現在のファイル名	新しいファイル名
【ファイルの削除】	del	削除するファイル名	
【フォルダの作成】	mkdir	作成するフォルダ名	

なお、MacやLinuxの場合は、上から順にそれぞれ「cp」「mv」「rm」「mkdir」コマンドを代わりに使ってください。

```
C:¥Users¥sugawara>mkdir myfolder ↵

C:¥Users¥sugawara>copy Main.java myfolder¥Main.java ↵
1 個のファイルをコピーしました。
C:¥Users¥sugawara>cd myfolder ↵
C:¥Users¥sugawara¥myfolder>ren Main.java Sub.java ↵
C:¥Users¥sugawara¥myfolder>del Sub.java ↵
```

これらファイルの操作については「マイコンピュータ」や「エクスプローラ」などからマウスで操作しても構わないよ。

JREとは

購入したばかりのパソコンで、「javacコマンドは使えなくてもjavaコマンドは動作する」ことがあります。これはそのコンピュータにJDKではなく**JRE**(Java Runtime Environment)が導入されているからです。

JREはJDK同様、オラクル社などがWebサイトで無償で公開しているJavaのダウンロードパッケージです。「自分ではプログラムを開発しないが、他人が開発したプログラムを実行する人」のためのものであるため、javaコマンドは含まれていますがjavacコマンドは含まれません。

A.3 ソース編集・コンパイル・実行

A.3.1 JDK を用いた開発の全体像

JDK を用いた開発の全体像をまとめたものが、次の図です。

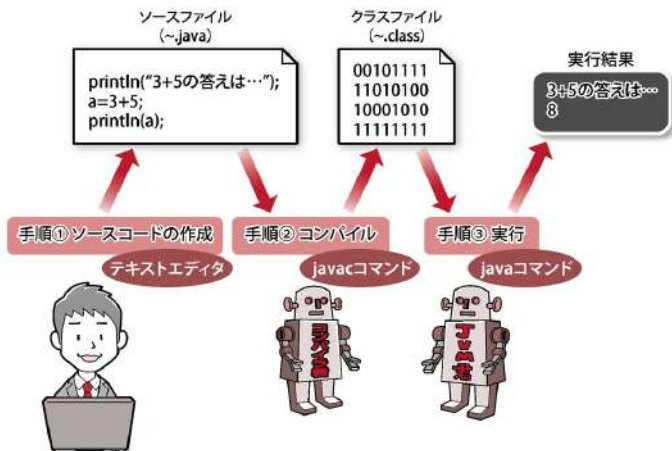



図 A-2　JDK を用いた開発の全体像

まず開発者は、テキストエディタを使ってソースコードを作成します。ソースコードは、クラス名に拡張子「.java」を付けて（クラス名が「Main」なら「Main.java」となる）保存します。

次に、コマンドラインプロンプトから javac コマンドを使って Java コンパイラを起動します。Java コンパイラは、指定されたソースファイルを実行可能な**クラスファイル**（class file）に変換します。そして、java コマンドでインタプリタを起動すると、クラスファイルの中身が JVM に読み込まれて実行されます。

バイトコードと仮想マシン

Windows や Mac などパソコン用のプログラムは、スーパーコンピュータ（スパコン）では動きません。なぜなら、パソコンとスパコンでは、CPU が理解できる命令（マシン語）が異なるからです。

しかし、第 0 章の冒頭で触れたように、Java で開発したプログラムはパソコンでもスパコンでも同じように動きます。これは、javac コマンドで出力されるバイトコードが、特定の CPU に依存しない汎用的なマシン語であるためです。Windows パソコンで実行する際には Java のインタプリタによって Windows 用のマシン語に、スパコンで実行する場合にはスパコン用のマシン語に変換されます。

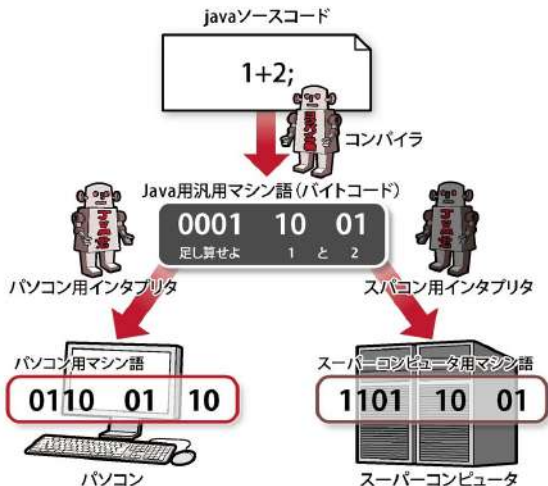



図 A-3　Java コンパイラと Java インタプリタの処理

見方を変えれば、パソコンやスパコンの中にある「汎用マシン語を理解するコンピュータ」がバイトコードを実行しているようにみえます。これが、「Java 仮想マシン（JVM）」という名前の由来です。

A.3.2 手順① ソースコードの作成

①-1 プログラム開発用のフォルダの作成

新しいプログラムを作成する場合、ソースファイルやクラスファイルを入れるための新しいフォルダを作成します。名前は何でも構いませんが、新しいプログラムを作成するたびに名前を変えた別のフォルダを作ることをお薦めします。複数のプログラム用のファイルを1つのフォルダに入れてしまうと、どのファイルがどのプログラムのものかわからなくなったり、ファイル名が衝突したりするからです。

①-2 エディタを起動してソースコードを入力する

p.210 で解説したように、テキストエディタを使用してソースコードを編集します。Microsoft Word などのワープロソフトは、テキストではなく独自形式で文書を保存するので、ソースコードの入力には使えません。

①-3 ソースファイルを保存する

ソースコードの作成が終わったら、①-1 で作成したプログラム開発用フォルダに**ソースファイル**として保存します。このとき、ファイル名は必ず「(クラス名).java」という名前にする必要があります。たとえば Main クラスを書いたソースコードであれば、「Main.java」となります。

A.3.3 手順② コンパイルする

②-1 作業フォルダへの移動

コマンドプロンプトを起動し、cd コマンドで開発用フォルダに移動します。dir コマンドを使って、先ほど作成したソースファイルが存在することを確認します。

②-2 コンパイルの実行

コンパイルは、「javac（ソースコードファイル名）」で実行します。たとえば、Main.javaをコンパイルしたい場合には次のように入力します。

```
C:¥Users¥sugawara¥java¥hello>javac Main.java ↵
C:¥Users¥sugawara¥java¥hello>
```

上記の画面出力例のように、javacコマンドを実行しても何も表示されずに次のプロンプトが出てきた場合、コンパイルは成功しています。逆に、もしソースコードに間違いがあった場合は、次のようにコンパイルエラーが表示されます。

```
C:¥Users¥sugawara¥java¥hello>javac Main.java ↵
Main.java:3: 文字列リテラルが閉じられていません。
System.out.println("Hello World);
                   ^
Main.java:3: ';' がありません。
```

エラーとして指摘された部分をエディタで修正し、エラーがなくなるまでコンパイルを繰り返します。

なかなかエラーがなくならないときや、エラーの原因がわからない場合の対処方法を本書の付録Cにまとめてある。ぜひ活用してほしい。

A.3.4 手順③ プログラムの実行

③-1 クラスファイルの確認

コマンドプロンプトで開発用フォルダに移動し、コンパイル結果のクラスファイルが存在することを確認します。

```
C:¥Users¥sugawara¥java¥hello>dir ↵
    ⋮
```

```
2011/08/15 15:56 413 Main.class
2011/08/15 12:49 106 Main.java
    ⋮
```

クラスファイル

③ -2 クラスファイルの実行

実行は、「java（クラスファイル名から .class を取り除いたもの）」で行います。たとえば、Main.class を実行する場合は、「java Main」と入力します。

```
C:¥Users¥sugawara¥java¥hello>java Main ↵
Hello World
```

コンパイルするときには .java を付けるのに、実行するときは .class を付けないのね。

そう、**実行の際には .class は付けない**、これが Java のルールなんだ。

次の第 6 章では複数のクラスを使った Java プログラミングについて解説していきます。第 6 章に登場するプログラムは dokojava では実行することができないので、本章で学んだ javac と java コマンドで学習を進めてください。

なお、以降のコマンドプロンプト利用の解説では、特段の理由がある場合を除いて、プロンプトは単に > とだけ表記します。

第6章

複数クラスを用いた開発

第 5 章で学習したメソッドを上手に使えば、
ある程度大きなプログラムも 1 人で作ることができます。
しかし大規模なソフトウェアの開発になると、
自分以外の開発者と手分けしてプログラミングする必要があります。
この章では、複数の開発者が分担して部品を作り、それを組み合わせる
Java のしくみを紹介します。

CONTENTS

6.1 ソースファイルを分割する

6.1.1 1 つのソースファイルによる開発の限界

菅原さん！　はりきって開発をしていたら、クラスの中に 35 個もメソッドができてしまって、ワケがわかんなくなっちゃいました！

そんなときにはクラスを分割すればスッキリするよ。

第 5 章では、長く複雑になってしまった main メソッドを複数のメソッドに分割する方法を学びました。しかし、1 つのソースファイルの中に含まれるメソッドの数が増えると、やはりソースコードの全体を把握することが難しくなり、開発しにくくなっていきます。

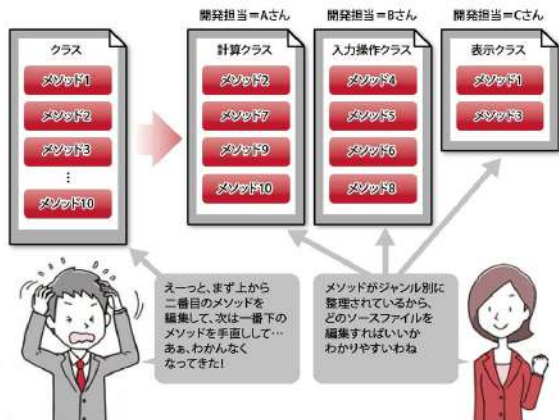



図 6-1　複数の開発者が分担して開発するための便利なしくみ

そこでJavaでは、1つのソースファイルにすべてのメソッドを書くのではなく、複数のソースファイルに分割して記述できるようなしくみが準備されています。

1つのソースファイルにはクラスブロックを最低1つは作成しなければならないルールがありますので、**複数のソースファイルに分けて開発するということは、複数のクラスに分けて開発すること**だと捉えることもできます。

たくさんのメソッドを複数のクラスに分けて記述することには、単に「整理されてわかりやすくなる」だけではなく、「**ファイルごとに開発を分担し、それぞれが並行して開発を進められる（＝分業しやすい）**」というメリットもあります。このように、1つのプログラムを複数の部品に分けることを**部品化**といいます。

6.1.2 計算機プログラムを分割しよう

では、リスト6-1の計算機プログラム（Calc）を、2つのクラスに分割してみましょう。現状の計算機プログラムはmain()、tasu()、hiku()の3つのメソッドから構成されています。

リスト6-1 計算機プログラム

Calc.java

```
public class Calc {
  public static void main(String[] args) {
    int a = 10; int b = 2;
    int total = tasu(a, b);
    int delta = hiku(a, b);
    System.out.println("足すと" + total + "、引くと" + delta);
  }
  public static int tasu(int a, int b) {
    return (a + b);
  }
  public static int hiku(int a, int b) {
    return (a - b);
  }
}
```

さぁ、この3つのメソッドをのうち、どれを別クラスに切り出そうか。

tasu() と hiku() の2つは「数学的な計算処理をするメソッド」であり、main() は「tasu() や hiku() を呼び出して画面に表示する役割を持つ、全体の流れを司るメソッド」です。よって main() とそれ以外のメソッドを2つのクラスに分けて整理しましょう。

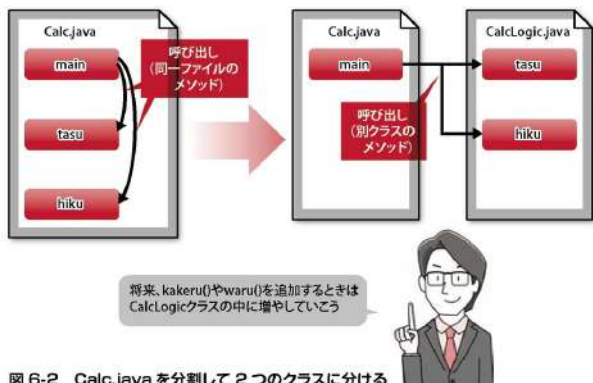



図 6-2　Calc.java を分割して2つのクラスに分ける

Step1　計算処理メソッドを記述するためのソースファイルを作る

まず、tasu() や hiku() といった計算ロジックのメソッドを入れるソースファイルを作ります。新たなファイル名は CalcLogic.java にします。1.2.1 項で「ソースファイル名とクラス名は同じでなければならない」と学びましたね。そのため、CalcLogic.java の書き始めは「public class CalcLogic」とします。

Step2　tasu() と hiku() を移動する

現在 Calc.java の中にある tasu() と hiku() を、新たに作った CalcLogic.java へ移動します。すると、CalcLogic.java は次のリスト 6-2 のようになります。

リスト 6-2　CalcLogic.java に計算処理を追加する

CalcLogic.java

```
1  public class CalcLogic {
2    public static int tasu(int a, int b) {
3      return (a + b);
4    }
5    public static int hiku(int a, int b) {
6      return (a - b);
7    }
8  }
```


Step3　メインメソッド内の呼び出しを修正する

一方で、Calc.java には次のように main() だけが残されているはずです。

リスト 6-3　Calc.java

Calc.java

```
1  public class Calc {
2    public static void main(String[] args) {
3      int a = 10; int b = 2;
4      int total = tasu(a, b);
5      int delta = hiku(a, b);
6      System.out.println("足すと" + total + "、引くと" + delta);
7    }
8  }
```

4 行目と 5 行目で tasu() や hiku() メソッドを呼んでいますが、このままでは「tasu() や hiku() メソッドがないから呼び出せない！」という意味のコンパイルエラーが出てしまいます。**この Calc.java には tasu() や hiku() は存在しない**ので当然です。

今まで、main()の中で単に「tasu(a,b)」と記述すればtasu()を呼び出すことができたのは、**main()とtasu()やhiku()が同じCalcクラスに属していたから**です。しかし今回のソースファイルの分割によって、tasu()やhiku()はCalcLogicクラスに属するようになったため、main()から呼び出す際には「**CalcLogicのtasu()**」や「**CalcLogicのhiku()**」のように、明示的に所属を示す必要があります。これには、main()から以下のように呼び出すことで対応できます。

```
int total = CalcLogic.tasu(a, b);
int delta = CalcLogic.hiku(a, b);
```

ドット(.)は次の章以降でも頻繁に登場するけど、日本語で言う「～の」という意味だよ。

同じ部署の私たちは先輩を普段「菅原さん」と呼べるけど、別部署の人は「開発部の菅原さん」と言うのと似ていますね。

ここまでで無事、計算機プログラムは2つのクラス(リスト6-2、リスト6-4)に分割することができました。

リスト6-4　Calc.java

Calc.java

```
1  public class Calc {
2    public static void main(String[] args) {
3      int a = 10; int b = 2;
4      int total = CalcLogic.tasu(a, b);
5      int delta = CalcLogic.hiku(a, b);
6      System.out.println("足すと" + total + "、引くと" + delta);
7    }
8  }
```

6.2 複数クラスで構成されるプログラム

6.2.1 複数クラスのコンパイル

前節では計算機プログラムを2つのソースファイルに分割しました。そのため、このプログラムを実行するには、Calc.java と CalcLogic.java のそれぞれをコンパイルする必要があります。javac コマンドでは、以下のように複数のソースファイルを指定することができます。

```
>javac Calc.java CalcLogic.java⏎
```

無事コンパイルが終了すると、それぞれのソースファイルに対応したクラスファイルが生成されます。

```
>dir⏎
2011/06/23 15:52 735 Calc.class
2011/06/23 15:51 234 Calc.java
2011/06/23 15:52 298 CalcLogic.class
2011/06/23 15:51 156 CalcLogic.java
4 個のファイル 1,423 バイト
```

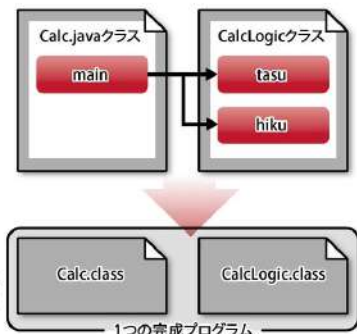



図 6-3　Calc.java の分割により、Calc.java と CalcLogic.java の 2 つで 1 つの完成プログラムを構成

6.2.2 Java プログラムの完成品

この 2 つのクラスファイルが計算機プログラムの最終完成品だ。誰かに渡す際には、この 2 つのクラスファイルが必要なんだよ。

えっ？　2 つのファイルを渡すんですか？ なんだか完成品じゃないような気がするんですけど…。

普段用いるパソコンのアプリケーションに慣れ親しんでいると、「Java プログラムの完成品」のちょっと変わった姿を意外に思うかもしれません。普通のプログラムは、たいていファイルは 1 つだけだからです。たとえば、Windows7 におけるメモ帳プログラムは、C:¥Windows¥System32¥notepad.exe のような単独のファイルであって、これをダブルクリックすれば起動します。

しかし、Java で開発されたプログラムは「複数のクラスファイルの集まり」であることが多く、ダブルクリックで起動させるのではなく、java コマンドで起動します。ですから Java プログラムを誰かに渡す、あるいは納品する場合には、**複数のクラスファイルが入っているフォルダをまるごと「1 つの完成品」として渡す**ことがほとんどです。

Java プログラムの完成品

- Java プログラムの完成品は、複数のクラスファイルの集合体。
- 誰かに配布する場合には、すべてのクラスファイルを渡す必要がある。

6.2.3 プログラムの実行方法

でも、「クラスファイルがたくさん入ったフォルダをまるごと」受け取ったら、どうやって起動すればいいのかしら？

JARファイル(コラム参照)でまとめた形ではなく、クラスファイルが入ったフォルダをまるごと受け取った場合は、クラス名を指定して実行する必要があります。

```
>java クラス名⏎
```

JVMは**起動時に指定されたクラスの中にあるmainメソッド**を呼び出してプログラムの実行を開始します。よって、Javaのプログラムを実行する人は**「渡された複数のクラスファイルのうち、mainメソッドが含まれているクラスの名前」を指定する必要**があります。 たとえば、計算機プログラムの場合は「java Calc」と起動すべきであって、「java CalcLogic」では正常に動作しません。

今回の計算機プログラムの場合、私たちは「Calcの中にmain()が入っていて、CalcLogicの中にはない」という事実を知っているので「java Calc」で起動できると判断できました。

しかし、他人が作ったJavaプログラムの場合は、すべての**クラスファイルを受け取っても、「どのクラスの中にmainメソッドがあるか」がわからないと起動できない**ことに注意しましょう。

複数の完成クラスファイルを渡す場合の注意点

Javaのプログラムを配布・納品するときは、単にすべてのクラスファイルを渡すだけではなく、「どのクラスにmain()が入っているか」も伝える必要がある。

JARファイルとは?

プログラムの完成品が複数のクラスファイルになった場合、メールで送る際などに不便です。そこでJavaでは、「複数のクラスファイルを1つにまとめるファイル形式」として**JAR**(Java ARchive)が定められています。JARファイルはZIPファイルととてもよく似たアーカイブファイルであり、JDKに付属するjarコマンドでも作成することができます。

パッケージを利用する

6.3.1 クラスが増えすぎたら…どうしよう？

菅原さ〜ん！

今度はクラスの数が増えすぎて、わかりにくくなっちゃった…だね？

Java を学習し始めて日も浅い段階では想像がつかないかもしれませんが、大規模なプロジェクトになると、数百個ものクラスを使って 1 つのプログラムを開発することもあります。しかし、クラスの数も 20 個を超える規模になると、さすがに管理が大変になってきます。

そこで Java には、各クラスを**パッケージ**（package）というグループに所属させて、分類・管理できるようなしくみが準備されています。

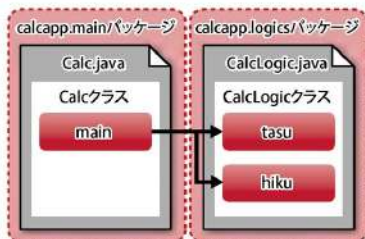



図 6-4　2 つのパッケージに分割された計算機プログラム。calcapp.main パッケージと calcapp.logics パッケージで構成

main() の行数が増えたら複数メソッドに分割 → メソッド数が増えたら複数クラスに分割 → クラス数が増えたら複数パッケージに分割というわけね。

Java には部品化のしくみがいくつも準備されているんだね。

それでは、前節でも登場した計算機プログラムを題材にして、パッケージを利用してみましょう。クラスをパッケージに所属させるためには、そのクラスの**ソースコードの先頭に package 文を記述**します。

クラスをパッケージに所属させる

package 所属させたいパッケージ名;

※ package 文はソースコードの先頭に記載する必要がある。

たとえば計算機プログラムの2つのクラスを、図6-4のようにそれぞれのパッケージに所属させるには、次のように記述します。

リスト6-5　Calc クラスを calcapp.main パッケージに所属させる

Calc.java

```
package calcapp.main;
public class Calc {
 ⋮
```

※ このコードは、後の6.7節で紹介する方法を用いて実行する必要があります。現状ではコンパイルまでできれば構いません。

リスト6-6　CalcLogic クラスを calcapp.logics パッケージに所属させる

CalcLogic.java

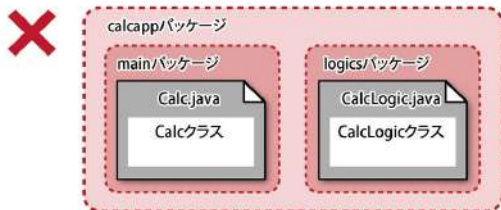
```
package calcapp.logics;
public class CalcLogic {
 ⋮
```

※ このコードは、後の6.7節で紹介する方法を用いて実行する必要があります。現状ではコンパイルまでできれば構いません。

パッケージの名前は、Javaの識別子（1.3.2項）のルールに従っていれば自由に定めることができますが、アルファベットは小文字を使用するのが一般的です。また、「calcapp.main」や「calcapp.logics」のように、ドットで区切ったパッケージ名も多く用いられます。

なお、「calcapp.main」と「calcapp.logics」という2つのパッケージ名を見て、「共通の calcapp パッケージに所属する main と logics という子パッケージで、同じ

グループである」という感覚を抱いてしまうかもしれませんが、両者は相互にまったく関係がない、独立した 2 つのパッケージです。パッケージの中にパッケージを入れることはできませんし、**パッケージに親子関係や階層関係はありません。**

正しくないパッケージのイメージ

正しいパッケージのイメージ

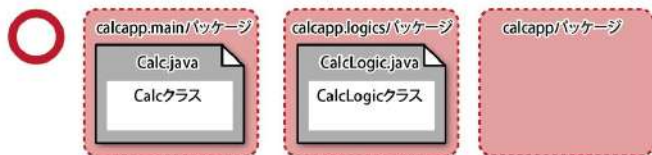



図 6-5　パッケージ名の一部が同じであっても、それぞれのパッケージに関連性はない

デフォルトパッケージ

前節まで作成してきたクラスには package 文がなく、どのパッケージにも所属していませんでした。どのパッケージにも所属していないことを「**無名パッケージ**に属している」または「**デフォルトパッケージ**に属している」と表現することもあります。

なお、デフォルトパッケージに属するクラスは後述の import 文でインポートすることはできません。

6.3.2 パッケージを含むクラス名を指定する

ここまでで無事2つのクラスを別パッケージに所属させることができました。しかし、このままコンパイルするとCalc.javaの2つの行に構文エラーが発生してしまいます。

```
int total = CalcLogic.tasu(a, b);
int delta = CalcLogic.hiku(a, b);
```

Calcクラスの中にあるこの2行では、それぞれ「CalcLogic」クラスを利用しようとしています。しかし、この書き方では**「どのパッケージのCalcLogicクラスか」を明示していない**ため、Calcクラスは自分と同じパッケージ(calcapp.mainパッケージ)に所属するCalcLogicクラスを呼び出そうとして失敗してしまうのです。別パッケージに所属しているクラスを利用する場合、次のように**所属パッケージ名を添えたクラス名**を指定する必要があります。

リスト6-7　別のパッケージにあるクラスを呼び出す

Calc.java

```
1  package calcapp.main;
2  public class Calc {
3    public static void main(String[] args) {
4      int a = 10; int b = 2;
5      int total = calcapp.logics.CalcLogic.tasu(a, b);
6      int delta = calcapp.logics.CalcLogic.hiku(a, b);
7      System.out.println("足すと" + total + "、引くと" + delta);
8    }
9  }
```

※このコードは、後の6.7節で紹介する方法を用いて実行する必要があります。現状ではコンパイルまでできれば構いません。

クラス名の前に所属パッケージ名を付けてあげればいいんですね。

厳密に「calcapp.logicsパッケージ」の「CalcLogicクラス」の「tasu()」と指定するんだよ。

このように、あるクラスから別パッケージのクラスを利用する場合、「パッケージ名を頭に付けた完全なクラス名」を使う必要があります。この完全なクラス名のことを、完全限定クラス名（full qualified class name）、または略してFQCNといいます。

完全限定クラス名（FQCN）

パッケージ名.クラス名

あまり一般的ではないけど、「同じパッケージに所属する別のクラス」を利用するときに、わざわざFQCNを使っても文法違反にはならないんだ。

同じ部署にいる私が菅原さんのことを「ミヤビリンクの、開発部の、菅原さん」と呼んでも、一応間違いではないのと同じですね。

そうだね。でも実際に社内でそんな呼ばれ方をしたら、「熱でもあるんじゃないのか？」と心配するよ（笑）。

6.4 名前空間

6.4.1 パッケージを使うもう 1 つのメリット

パッケージには「クラスをグループ化して分類・整理することで、プログラムをわかりやすくする」という目的のほかに、もう 1 つ重要な役割があります。それは、**自分が作るクラスに対して、開発者が自由な名前を付けられるようにすること**です。

え？ 今までも自由にクラス名を付けてきたし、不自由は感じませんでしたけど…。

そうだね。では、200 個ぐらいのクラスを 20 人で分担して開発する場合を考えてみようか。

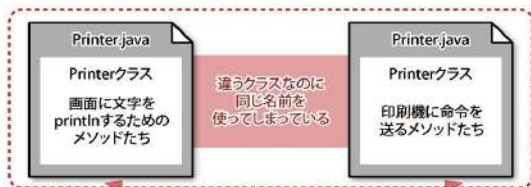
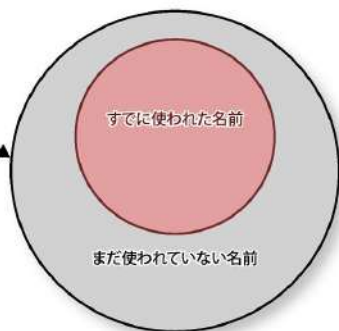



図 6-6　別のクラスに同じ名前を付けてはダメ

大規模な開発になると、複数の開発者が分担して各自が受け持ったクラスを開発します。すると、それぞれの開発者が偶然「同じクラス名を使ってしまう」可能性が出てきます（図6-6）。

このように、内容が異なる別々のクラスで同じ名前を取り合ってしまうことを**名前の衝突**といいます。

図6-7 新しいクラスを作るごとに、使えるクラス名が減っていく

異なるクラスで同じクラス名を使うと区別が付かなくなってしまうため、Javaではクラス名の衝突は原則として許されません。使うことができる名前の総量（=**名前空間**）は限られていて、**新しくクラスを作ると、そのクラス名は使えなくなり、使えるクラス名は減っていく**のです（図6-7）。

現実世界でも、新しく子どもが産まれて名前を付けるとき、もし「過去に使われた名前はダメ」という規則があったとしたら大変だろう？

そうですよね…。でも、どうして現実世界では名前が衝突しても問題ないんだろう？

現実世界で人名が重複しても問題が起きないのは、「他の手段によって正しく区別できる」からです。たとえば同じ会社に同姓同名の人がいたとしても、「部署」や「役職」などによって区別がつきます。

Javaでも**パッケージが異なれば、同じクラス名を使ってよい**というルールになっています。なぜならクラス名が同一でも、パッケージ名が異なれば**完全限定クラス名（FQCN）が異なるので両者を区別できる**からです。

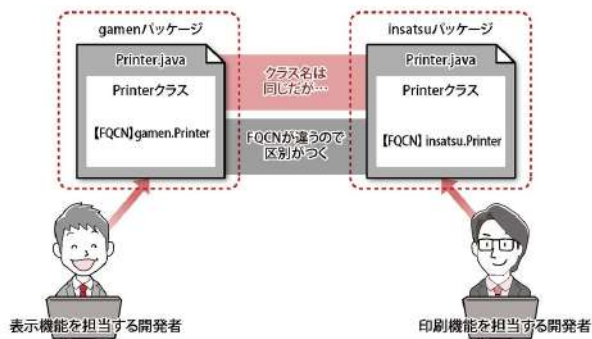



図 6-8　パッケージ名が異なれば完全限定クラス名（FQCN）が異なるので同じクラス名でも区別がつく

つまり**パッケージを使うことによって、それぞれのパッケージ内では自由にクラス名を付けることが可能になる**わけです。

6.4.2　パッケージ名自体の衝突を避ける方法

でも、パッケージ名が衝突しちゃうと困るんじゃないですか？

そうなんだ。だからパッケージ名の付け方については「あるルール」が推奨されているんだよ。

パッケージ名さえ異なればクラス名は重複してもよく、自由にクラス名を付けても構わないことがわかりました。しかし、**パッケージ名が衝突すると、これらの前提はすべて崩れてしまいます。**

自社の開発プロジェクトであれば、誰がどのようなパッケージ名を使うかを事前に決めておくことで衝突を回避できます。しかし、他社のパッケージを利用する際にはどうでしょう？ パッケージ名が衝突しないように事前に摺り合わせするのは困難です。

たとえば、A 社が myapp パッケージを使ってプログラムを開発しているとしましょう。画面表示を担当する Printer クラスは A 社内で開発しましたが、印刷

機能はイギリスの C 社が無料でインターネット上に公開している Printer クラスを利用すれば A 社内で開発せずに済みそうです。

しかし、C 社も偶然そのプログラムで myapp パッケージを使っており、このままでは A 社が作成した myapp.Printer と完全限定クラス名が重なってしまいます。これでは 2 つのクラスを区別できません。

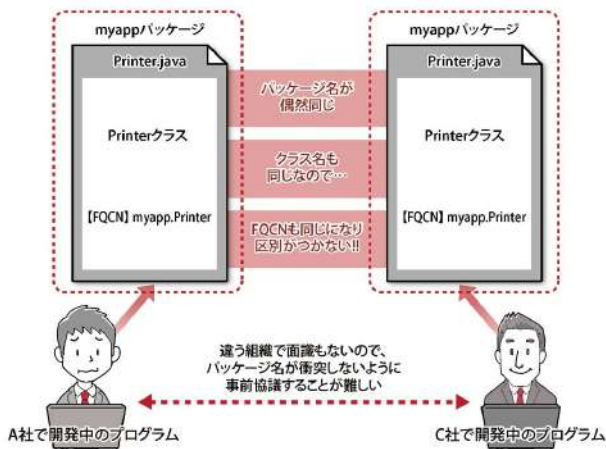



図 6-9　まったく面識がない開発者とはパッケージ名とクラス名の調整ができない!

そこで Java では、次のようなパッケージ名を用いることを推奨しています。

推奨されるパッケージ名

パッケージ名は、「保有するインターネットドメインを前後逆順にしたもの」から始める。

たとえば、foo.example.com というインターネットドメインを取得している企業であれば、com.example.foo で始まるパッケージ名を使うということです。イ

ンターネットドメインは世界に 1 つだけですから、これでパッケージ名が衝突することはありません。com.example.foo より後は、企業や組織内部でパッケージ名が衝突しないよう調整を行えばよいのです。

えっと…会社のホームページが http://miyabilink.jp/ だから…。

jp.miyabilink. から始まるパッケージ名を使えばいいんだよ。

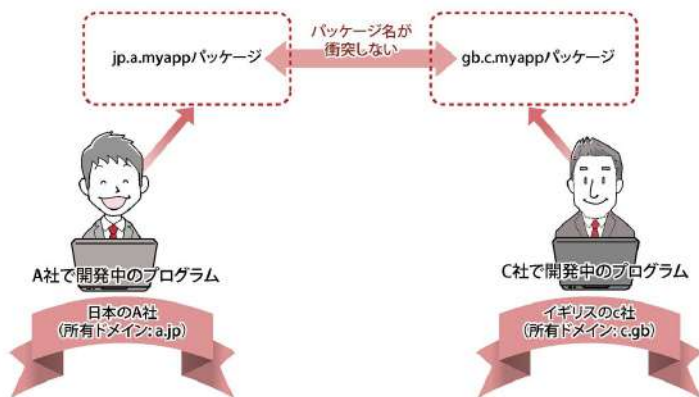



図 6-10　世界に唯一のインターネットドメインをパッケージ名に使うことで、パッケージ名の衝突を心配する必要がなくなる

このパッケージ名のルールのおかげで、自社だけではなく世界中のいろんな人や企業が作ったクラスを自由に組み合わせて利用できるようになるんですね。

そうだよ。Java のプログラムは「世界中の、さまざまな人が作ったクラス」を組み合わせることで効率よく作っていけるんだ。

テストプログラム等では簡単なパッケージ名を付けていても構いませんが、正式なプログラムのクラスには今回紹介した命名規則に従ったパッケージ名を付けましょう。あなたが作って公開するクラスも、世界中の誰かが利用する日が来るかもしれません（本書のサンプルコードでは、解説を簡単にするために、以後も簡単なパッケージ名を使います）。

6.4.3 完全限定クラス名の入力を省略する

名前空間のメリットはわかりましたけど、やっぱりパッケージ名の入力がめんどうですよ…。

めんどうくさがりな君にピッタリの構文があるよ（笑）。

再度、リスト6-7（p.233）のCalc.javaを見て、FQCNを利用している部分を確認してください。

Calc.java

```
1 public class Calc {
2   public static void main(String[] args) {
3     ⋮
4     int total = calcapp.logics.CalcLogic.tasu(a, b);
5     int delta = calcapp.logics.CalcLogic.hiku(a, b);
6     ⋮
7   }
8 }
```

FQCNの利用

「calcapp.logics.CalcLogic」という長い完全限定クラス名（FQCN）を2か所に記述しています。現時点では2か所で済んでいますが、将来プログラムが大きくなったら、この長いFQCNを何度もプログラムコードの随所に入力する必要が出てくるかもしれません。このような場合は、**import文**を使うことによって、FQCN入力のめんどうさを軽減できます。

FQCN入力の手間を省くための宣言

importパッケージ名.クラス名;

※import文はソースコードの先頭に、ただしpackage文より後に記述する。

では、Calc.javaのpackage文の下にimport文を記述してみましょう。

リスト6-8　Calc.javaにimport文を追加する

Calc.java

```
1  package calcapp.main;
2  import calcapp.logics.CalcLogic;
3  public class Calc {
4    public static void main(String[] args) {
5      ⋮
6      int total = CalcLogic.tasu(a, b);
7      int delta = calcapp.logics.CalcLogic.hiku(a, b);
8      ⋮
9    }
10 }
```

FQCNでなくてもエラーにならない

FQCNを指定してもよい

※このコードは、後の6.7節で紹介する方法を用いて実行する必要があります。現状ではコンパイルまでできれば構いません。

2行目のimport文に注目してください。この文は、「このソースコードで**単にCalcLogicという表記があったら、それはcalcapp.logics.CalcLogicのことだ**と解釈しなさい」という指示です。頻繁に利用するクラスはimport文を使ってインポートしておくことによって、完全限定クラス名を毎回指定する必要がなくなります。

もし、calcapp.logicsパッケージに所属するすべてのクラスをインポートしたい場合には、次のような記述も可能です。

リスト6-9　calcapp.logicsパッケージに属するすべてのクラスをインポート

Calc.java

```
1  package calcapp.main;
```

```
2  import calcapp.logics.*;
3  public class Calc {
4    ⋮
5  }
```

ただし、「import calcapp.*;」という記述では calcapp.main と calcapp.logics に所属するすべてのクラスを一度にインポートできないことに注意してください。なぜなら 6.3.1 項の図 6-5 にあるように、「calcapp.main」と「calcapp.logics」そして「calcapp」はまったく異なるパッケージであり、親子の関係にないためです。

この指定では、calcapp パッケージに所属する全クラスのみがインポートされるのであり、calcapp.main と calcapp.logics に所属するすべてのクラスをインポートしたい場合には以下のように記述する必要があります。

```
import calcapp.main.*;
import calcapp.logics.*;
```

import 宣言はあくまでも「めんどうさ軽減機能」

Java 以外のプログラミング言語の中には、「include 命令」や「require 命令」といったものでファイル名を指定することで、他のファイルに記述された機能が利用可能になる言語があります。ときどき Java の import 文も、include 命令や require 命令のようなものだと勘違いされてしまうことがありますので注意してください。

Java では**いっさいの宣言をすることなく、JVM が扱えるすべてのクラスを常時使うことができます**。ただし、その利用に際しては必ず FQCN を利用しなければならず、**import 文はあくまで FQCN の記述を省略してめんどうを軽減するため（開発者がラクをするため）の構文にすぎません**。import したからといって利用できるクラスやメソッドが増えたり、プログラムから利用できる機能が増えたりするようなことはないのです。

6.5 Java APIについて学ぶ

6.5.1 世界中の人々の協力で完成していたHelloWorld

「パッケージの命名規則を守れば、世界中のいろんな人が作ったクラスと自分のクラスを一緒に動かせる（6.4.2項）」ってことでしたけど…ちょっと想像できません。

そうですよ…ボクが作るプログラムなんて、しょせん社内の数人で作るものばかりで、世界をまたにかけた開発だなんて、そんな大げさな…。

何を言っているんだ。君たちは、この本の冒頭から「世界をまたにかけたプログラム」を作ってきたじゃないか。

この本の冒頭で私たちが初めて開発したプログラムは「HelloWorld」でした。それは画面に文字を表示するだけの、クラスを1つしか作らない、とてもシンプルなプログラムでしたね。

しかし、このHelloWorldプログラム、**実は1つのクラスだけでできているプログラムではありません**。私たちが作成したクラスは1つだけですが、実際には多くのクラスから成り立っています。試しに、javaコマンドに特殊なオプションを指定してHelloWorldプログラムを実行してみます。

```
>java -verbose:class HelloWorld↵
[Opened ...rt.jar]
[Loaded java.lang.Object from ...]
[Loaded java.io.Serializable from ...]
[Loaded java.lang.Comparable from ...]
```

```
[Loaded java.lang.CharSequence from ...]
[Loaded java.lang.String from ...]
 ⋮
 ⋮
[Loaded HelloWorld from ...]
 ⋮
 ⋮
```

実行する環境や JVM のバージョンによって多少の違いはありますが、著者の環境では 348 行の「Loaded ～.～.～」が表示されました。これらの行に表示される内容は、HelloWorld プログラムが動作するために JVM に読み込まれたクラスの完全限定クラス名です。

つまり HelloWorld プログラムとは、「**私たちが作った 1 つのクラスが、ほかの 347 個のクラスと連携して動く、計 348 クラスからなるプログラム**」だったわけです。

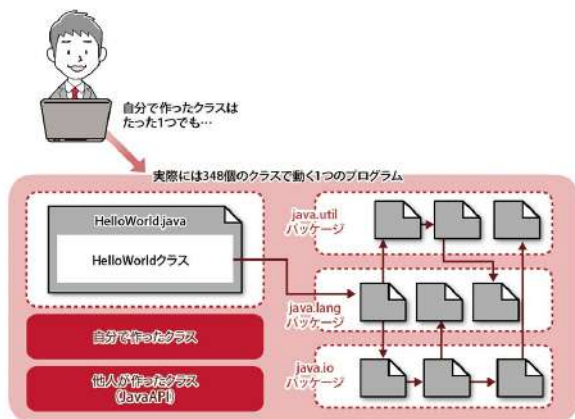



図 6-11　自分で作った Java のクラスが 1 つだけだったとしても、数百個の JavaAPI クラスが読み込まれて動作する

私たちが作った 1 つのクラスを除く 347 個のクラスは、Java に初めから標準添付されているクラスであり、それらは **API**（Application Programing Interface）

と総称されます。

Java では API として、およそ 200 を越えるパッケージ、3,500 を越える多くのクラスが標準提供されていて、私たちプログラム開発者は、それらのクラスをいつでも自由に利用することができます。

たとえば、「5 つの要素を持つ int 型配列」に入っている 5 つの整数を並び替えるプログラムを開発する場面を想定してみます。並び替えのロジックを自力で開発するのは少し大変ですが、わざわざ自分たちで開発しなくても、API として準備されている命令を呼び出せばすぐさま解決できるのです。

リスト 6-10　API 利用の例

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int[] heights = { 172, 149, 152, 191, 155 };
4      java.util.Arrays.sort(heights);
5      for (int h : heights) {
6        System.out.println(h);
7      }
8    }
9  }
```

Java が備える並べ替え命令

たった一行で並び替え完了なんて、なんてラクチンなんだ！

しかもラクなだけじゃない。この API は数学の専門家が作ったものだから、自分で作るより高速で動くし、バグもないんだよ。

本章まで学んできた今のみなさんであれば、このコードが「java.util パッケージの Arrays クラスにある sort メソッド」を呼び出していること、そして「java.util.Arrays は Java が標準で提供する API の一部であること」をすぐに理解できるでしょう。

実際、API に含まれる 3,500 個を越えるクラスは、それぞれクラスファイル（A

rrays.class など）の形で、JDK をインストールした際にコンピュータにコピーされています。これらのクラスファイルも、みなさんが HelloWorld クラスを作ったときと同じように、Java 言語を作った人たち（その多くが日本国外の技術者）がソースコードを書き、コンパイルして作ったものです。

みなさんは、自分でも気づかないうちに「世界中の人たちが作った 347 個のクラスと連携して動くクラスを作り、それを動かす」という、世界をまたにかけた開発をしていたのです。何ともスケールの大きい話だと思いませんか？

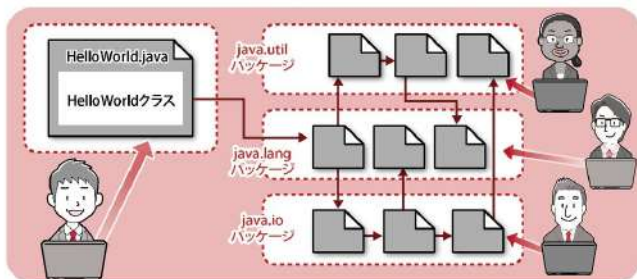



図 6-12　Java API に含まれる 3,500 個以上のクラスは、世界中の開発者が作り出したものである

6.5.2　API で提供されるパッケージ

すでに紹介したように、API には非常にたくさんのパッケージとクラスが含まれていますが、API のクラスには「java.」または「javax.」で始まるパッケージ名が利用されています。以下は代表的な API パッケージです。

表 6-1　Java API に含まれる代表的なパッケージ

java.lang	Java に欠かせない、重要なクラス群
java.util	プログラミングを便利にするさまざまなクラス群
java.math	数学に関するクラス群
java.net	ネットワーク通信などを行うためのクラス群
java.io	ファイル読み書きなど、データを逐次処理するためのクラス群

特に、java.lang パッケージに属するクラスは頻繁に利用するものが多いので、

「特に import 文を記述しなくても自動的にインポートされる」という特別扱いを受けることになっています。java.lang パッケージに属する代表的なクラスとしては、System、Integer、Math、Object、String、Runtime などがあります。

今までずっと画面への表示で使ってきた「System.out.println()」の System は、実は「java.lang.System」クラスだったんだよ。

6.5.3 API リファレンスの読み方

API には、どんなクラスが含まれているんですか？

呼び出すだけで簡単にゲームが作れちゃうようなクラスとかがあったらいいな。

Java が提供してくれている膨大な数の API クラスには、実際にどのようなクラスが含まれていて、どのようなメソッドを持っているか、興味がわいてくるかもしれませんね。

それを調べるためには、**API リファレンス**（API reference）と呼ばれる API の説明書を読む必要があります。説明書といっても、紙に印刷されたものではありません。Web ページでおなじみの HTML で書かれているファイルで、Web ブラウザを用いて閲覧します。

JDK をダウンロードしたオラクル社のサイトからファイルとしてダウンロードすることも可能ですが、そのままブラウザで閲覧することも可能です。検索サイトで、「Java　API　リファレンス」などのキーワードで検索することで API リファレンスのページに到達できるでしょう。

図 6-13 のように API リファレンスはフレームによって 3 分割されています。

図 6-13　Java の API リファレンスの画面

APIに含まれるあるクラスについて調べたい場合には、画面を次のような手順で操作します。

①左上のフレームで、調べたいクラスが所属するパッケージ名をクリックする。
②左下のフレームで、調べたいクラス名をクリックする。
③右側のフレームに表示されるクラスの説明を読む。

クラスの説明には、概要の説明のほか、そのクラスが持つメソッドやその引数・戻り値の一覧などが詳細に解説されています。

試しにjava.langパッケージのMathクラスを調べてみてください。解説を見ると、randomというメソッドを持っていることや、ほかにも数多くのメソッドを持っていることがわかります。

もっとAPIを知りたくなったら

Javaは、以下のように他にもたくさんのAPIを備えています。特によく用いるAPIの利用方法については、本書の続編『スッキリわかるJava入門 実践編 第2版』（インプレス）で紹介しています。

- 文字列の比較、照合、編集を行うAPI
- 情報をまとめて格納するAPI（コレクション）
- ファイルを読み書きするAPI
- ネットワーク通信を行うAPI
- データベースアクセスを行うAPI
- 複数の処理を同時実行するAPI（スレッド）

6.6 クラスが読み込まれるしくみ

6.6.1 必要なときに、必要な分だけ

私たちの HelloWorld を実行したら、裏で 347 もの API クラスが読み込まれて動いていたのには少し感動しました。でも、なぜ 3,500 以上もある API クラスのうち 347 個だけが読み込まれたんでしょうか？

そうだよね。プログラムが動いている途中に「やっぱり別の API クラスも必要になっちゃった」なんてことにはならないのかな？

前節の解説にあるように、1 つの Java プログラムが動くために、とても多くのクラスが JVM に読み込まれて動作します。このように、JVM が必要なクラスファイルを読み込む処理を**クラスローディング**（class loading）といいます。

API として提供されるクラスファイルは 3,500 個を超えますが、JVM は起動

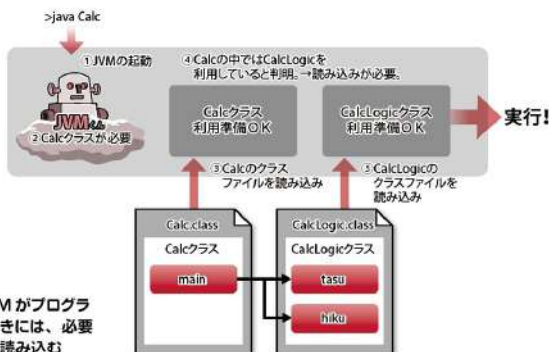



図 6-14　JVM がプログラムを実行するときには、必要なクラスを順次読み込む

直後にそのすべてを読み込むようなことはしません。使わないクラスまでロードしていたらムダにメモリを消費しますし、動作も遅くなってしまうからです。Java のクラスローディングのしくみは大変に賢く作られていて、一部の例外を除いて**「必要になったときに、必要なクラスだけ」を読み込む**ようになっています。

たとえば、Calc クラスを実行するときの状態を考えてみましょう。最初に JVM は Calc.class を読み込みます。そして、その内容を見て、「Calc クラスの中で CalcLogic クラスを呼び出す必要がある」ことに気づきます。そこで JVM は初めて CalcLogic.class ファイルを読みにいくのです。

もし Calc クラスの内部で CalcLogic クラスを使っている箇所をすべて削除した場合には、Calc クラスを実行しても CalcLogic クラスはロードされません。

6.6.2　クラス名だけでクラスファイルを探し出すためのしくみ

JVM の中でクラスファイルを読み込む仕事をしているのは、**クラスローダー**（class loader）という部分です。たとえば、JVM がクラスローダーに対して、「Calc クラスを利用するから、読み込んで利用可能にしなさい」という指示を出すと、クラスローダーはコンピュータのハードディスクの中にある Calc.class を読み込みます。

ここで着目してほしいのが、**JVM は使いたいクラス名を指定しているだけ**であって、**クラスファイルがハードディスクのどこのフォルダにあるのかをいっさい指定していない**点です。

図 6-15　JVM はクラスローダーに依頼してクラスを読み込んでもらう

Calc.class という目的のクラスファイルは、c:¥ にあるかもしれませんし、c:¥Program Files¥MyCalc¥lib にあるかもしれません。しかし、

クラスローダーは膨大な容量を持つハードディスクの中から一瞬で Calc.class ファイルを探し出して読み込んでくれます。

どうして一瞬で見つけられるんですか？　数百 GB もあるハードディスクを検索していたら、数秒…いえ数分はかかってしまいそうです。

確かに。でもクラスローダーはハードディスクの内容をすべて検索したりはせず、賢い方法で探し出すんだ。

クラスローダーは**クラスパス**（classpath）というヒント情報を使うことで、極めて高速に目的のクラスファイルを探し出します。クラスパスとは、「クラスローダーがクラスファイルを探す際に、見にいくべきフォルダの場所」のことで、あらかじめ 1 つ以上の場所を指定しておきます。

たとえばクラスパスとして「c:¥work」が指定してある場合、クラスローダーは c:¥work の中に Calc.class があるか探しにいくだけでよいため、高速に検索することができるのです。

6.6.3 クラスパスの指定方法

今までボクはクラスパスなんて指定してなかったですよ？

そうだね。そのタネあかしをしようか。

JVM が動作するときにクラスファイルを検索する場所であるクラスパスを指定するには、次の 3 つの方法があります。

方法 1：起動時に java コマンドで指定する

java コマンドで JVM を起動する際に、-cp オプションまたは -classpath オプションで指定する方法です。次のように指定します。

```
>java -cp c:¥work Calc⏎
```

方法 2：検索場所を OS に登録しておく

java コマンドを入力するたびに、いちいち -cp オプションを指定するのはめんどうですね。そこで、OS の「環境変数」という設定にクラスパスを登録しておくことができます。java コマンドは、この環境変数を自動的に読み込んでクラスファイルの検索に利用します。クラスパスは次の方法で登録します。

なお、環境変数の設定方法は OS によって異なりますので、詳細は OS のヘルプファイルや解説書を参照してください。

Windows の場合

①「コントロールパネル」-「システム」-「システムの詳細設定」-「詳細設定（タブ）」-「環境変数（ボタン）」の順にクリックする。
②ユーザー環境変数として CLASSPATH を追加し、値を設定する。

図 6-16　環境変数の設定

Mac や Linux の場合

（ユーザーのホームディレクトリ）/.profile というファイルの末尾に以下のような内容で追記する（/var/javadev をクラスパスとする場合）。

```
export CLASSPATH=/var/javadev
```

方法 3：特に指定しない

CLASSPATH 環境変数に指定がなく、-cp オプションの指定もない場合、デフォルトでは java コマンドが実行されたフォルダがクラスパスとなります。たとえば c:¥work で java コマンドを実行すれば、c:¥work がクラスパスに設定されます。

6.6.4 クラスパスで指定できる対象

クラスパスとして指定することができる対象は、次の3つから選べます。

対象1：フォルダの場所

クラスファイルが置かれているフォルダの場所（絶対パス）を指定するもので、最も一般的です。たとえば「c:¥work」という指定をすると、workフォルダ内のクラスファイルが検索対象となります。

対象2：クラスファイルが入ったJARファイルやZIPファイル

クラスファイルが入っているJARファイルやZIPファイルがあれば、そのファイルの場所（絶対パス）をクラスパスとして指定することができます。クラスローダーは指定されたファイルの中を検索し、もしクラスファイルが見つかれば読み込みます。

たとえば、Calc.classが入ったcalcapp.jarというファイルがc:¥work¥jarsにある場合、「c:¥work¥jars¥calcapp.jar」というクラスパス指定をすることで、Calc.classを読み込むことができるようになります。

対象3：複数のフォルダ、JAR／ZIPファイル、それらの組み合わせ

クラスパスとしては、複数のフォルダやJARファイル、ZIPファイルをデリミタ文字で区切って指定できます。デリミタ文字は、Windowsの場合はセミコロン（;）、LinuxやMacの場合はコロン（:）です。クラスローダーは、クラスファイルを探す際に指定された場所を前から順に探していきます。

Windowsの場合

```
c:¥work;c:¥work¥jars¥calcapp.jar
```

LinuxやMacの場合

```
/var/javadev:/var/javadev/jars/calcapp.jar
```

クラスパスで指定された場所以外にいくらクラスファイルを作っても、JVM はそのクラスを読み込めないんですね。

そのとおり。クラスは作ったのにプログラムがうまく起動できない場合、まずクラスパスを確認しよう。

クラスパスに自動的に加わる rt.jar

java コマンドでクラスファイルを実行する際には、特に指定しなくても rt.jar（または classes.jar）というファイルがクラスパスに追加されます。このファイルは JDK に含まれているもので、Java をインストールしたフォルダの配下にある lib フォルダの中に含まれています。

試しに、この JAR ファイルを ZIP ファイルの展開ツールで展開してみると、たくさんのフォルダとクラスファイルがあることがわかります。展開されたフォルダには java というフォルダがあり、その中には、lang というフォルダがあり、その中に System.class というクラスファイルがあり……。

そう、これらのファイルこそ膨大な数の API クラス群の実体なのです。rt.jar が自動的にクラスパスに加わるため、私たちは意識することなく System クラスなどの API を利用できていたのですね。

6.7 パッケージに属したクラスの実行方法

6.7.1 実行クラス名の正しい指定

おや？　そういえば、この章の前半で作ったCalcプログラム（p.241のリスト6-8）ですけど、コンパイルはできましたがエラーで実行できないですね。

こんなときは「逃げずに英語のエラーメッセージを読む」だったわね。

この章の前半では、package文を使ってクラスをパッケージに所属させる方法を学びました。CalcクラスとCalcLogicクラスはそれぞれ「calcapp.main」と「calcapp.logics」という別々のパッケージに所属させることができ、リスト6-8はコンパイルも正常に通ります。しかし、完成したCalcクラスをいざ実行しようとするとエラーに直面してしまいます。

```
>java Calc⏎
Exception in thread "main" java.lang.NoClassDefFoundError: Calc
Caused by: java.lang.ClassNotFoundException: Calc
  at java.net.URLClassLoader$1.run(Unknown Source)
  at java.security.AccessController.doPrivileged(Native Method)
  at java.net.URLClassLoader.findClass(Unknown Source)
  at java.lang.ClassLoader.loadClass(Unknown Source)
  at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
  at java.lang.ClassLoader.loadClass(Unknown Source)
```

「NoClassDefFoundError」は直訳すると「クラス定義が見つからない」というエラーです。実はこのプログラム実行方法には 2 つの問題点があるため、JVM は正しくプログラムを起動できていません。最初の問題点は、そもそも**起動しようとしているクラスの指定が誤っている**こと（図 6-17「問題点②」）です。

図 6-17　コンパイルはうまくいったのに、実行できない!?

初めて java コマンドを学習した際に、構文を以下のように紹介しました。

```
>java （ソースファイル名から.javaを取ったもの）
```

これまではこのような理解でも構いませんでしたが、パッケージを利用するようになった今、java コマンドのより正確な構文を理解する必要があります。

java コマンドの正確な構文

>java　起動したいクラスの完全限定クラス名（FQCN）

たとえば、計算機プログラムの場合は、次のように起動しなければなりません。

```
>java calcapp.main.Calc⏎
```

この入力により、JVM はクラスパスから「calcapp.main.Calc クラスの中身が格納されているクラスファイル」を自動的に探し出し、そのクラスファイルを読み込んで実行してくれます。

私たちはjavaコマンドを実行する際、「実行したいクラスがハードディスクのどこにあるのか」を指定する必要はありません。「どのクラスを実行したいか」だけを伝えてあげれば、あとはクラスローダーがクラスファイルを自動的に探し出してくれるのです。

「java Calc」ではデフォルトパッケージにあるはずのCalcクラスを実行しようとしてしまうんですね。

6.7.2 クラスファイルの正しい配置

先輩、やっぱり動きません。「Calcクラスが見つからない」っていうエラーが消えなくって…。

どうやらクラスローダーがCalc.classを見つけられていないみたいだね。

FQCNを指定してjavaコマンドを実行しても、まだエラーはなくなりません。実行すると次のようなエラーメッセージが表示されてしまいます。

```
>java calcapp.main.Calc⏎
Exception … java.lang.NoClassDefFoundError: calcapp/main/Calc
        ⋮
Could not find the main class: calcapp.main.Calc.  Program will exit.
```

最後の1行を和訳すると「calcapp.main.Calcというメインクラスが見つからなかった」という意味になります。どうやら、**クラスローダーが目的のクラスファイルを探し出せない**ようです。

前節で解説したとおり、クラスローダーは**クラスパスで指定されたフォルダを対象に**、探しているクラスファイルを調べます。実はこのとき、パッケージに属しているクラスを探す場合には、次のようなルールでクラスファイルを探すことになっています。

クラスローダーの動作原則

あるパッケージ x.y.z に属するクラス C を探す場合、クラスローダーは、「クラスパスで指定されたフォルダ ¥x¥y¥z¥C.class」というファイルを読み込もうとする。

つまり、パッケージに属したクラスファイルをクラスローダーに読み込んでもらうには、**現在のクラスパスを基準として、パッケージ階層に対応したフォルダ階層を作り、その中に必要なクラスファイルを配置しておく必要がある**のです。

たとえば c:¥work をクラスパスとする場合、コンパイルによって生成された Calc.class と CalcLogic.class は次のようなフォルダを作成し、その中に配置しておかなければなりません。

```
Calc.class         → c:¥work¥calcapp¥mainフォルダへ
CalcLogic.class    → c:¥work¥calcapp¥logicsフォルダへ
```

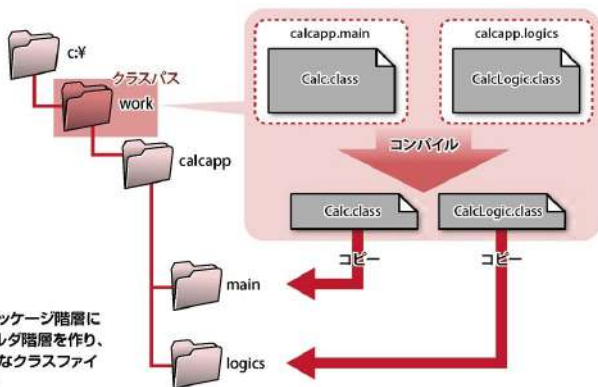



図 6-18　パッケージ階層に対応したフォルダ階層を作り、その中に必要なクラスファイルを配置する

クラスファイルを適切なフォルダに置いた上で実行すれば、次のような順序を経て、無事にプログラムは動作するでしょう。

① JVM は起動させるクラス名（calcapp.main.Calc）を受け取る。
② JVM はクラスローダーに対して calcapp.main.Calc を読み込むよう指示する。
③ クラスローダーはクラスパスを確認する。
④ クラスローダーは、クラスパスを基準として「calcapp」→「main」とフォルダを降りていき（すなわち、c:¥work¥calcapp¥main の中）、そこに Calc.class というファイルを発見する。
⑤ クラスローダーは発見した Calc.class を読み込む。
⑥ JVM は読み込み済みの Calc クラスの main メソッドを実行する。

やった！　やっと動きました！

お疲れさま。これで Java の基本は卒業だよ。文法も覚えたし、プログラムをメソッド・クラス・パッケージに部品化する方法もマスターしたね。

API の調べ方ももうわかるから、もうどんな大きなゲームだって作れるはずですよね！　よぉし、スゴいプログラムを作るぞ！

統合開発環境を用いた効率的な開発作業

この章では JDK を用いた開発を通してクラスパスについて学びました。ファイルの配置や起動時の指定など、少し複雑に感じた方もいるかもしれません。実際の開発現場では、より便利で効率的な作業のために「統合開発環境」と呼ばれる開発ツールを利用します。詳細は巻末の付録 B を参照してください。

第 6 章のまとめ

この章では、次のようなことを学びました。

クラスの分割

- 複数のクラスで 1 つのプログラムを構成することができる。
- 別のクラスのメソッドを呼び出す場合は、「クラス名.メソッド名」と指定する。
- Java プログラムの完成像は複数のクラスファイルの集合体である。
- 実行する際には、main メソッドが含まれるクラスの FQCN を指定して java コマンドを起動する。

パッケージ

- package 文を用いて、クラスをパッケージに所属させることができる。
- import 文を使うと、コード中の FQCN 指定を省略できる。

API

- Java にあらかじめ添付されている多数のクラス群を API という。
- API は通常「java.」や「javax.」で始まるパッケージ名を用いている。
- java.lang パッケージに属するクラスは自動的にインポートされる。
- API に用意されているクラスは、API リファレンスで調べることができる。

クラスローダーの動作

- クラスローダーは、読み込み対象クラスの FQCN に基づき、クラスパスを基準としてパッケージ階層に従ったフォルダ構成内を探し、読み込む。
- コンパイルして生成したクラスファイルは、実行時にクラスローダーが見つけられるように、適切なフォルダに配置しなければならない。

6.9 練習問題

練習 6-1

次のソースコードを３つのクラスに分割することを考えます。

Main.java

```
1  public class Main {
2    public static void main(String[] args) throws Exception {
4      doWarusa();        ┐
5      doTogame();        ┘ 前半
7      callDeae();        ┐
8      showMondokoro();   ┘ 後半
9    }
10   public static void doWarusa() {
11     System.out.println("きなこでござる。食えませんがの。");
12   }
13   public static void doTogame() {
14     System.out.println("この老いぼれの目はごまかせませんぞ。");
15   }
16   public static void callDeae(){
17     System.out.println
         ("ええい、こしゃくな。くせ者だ！であえい！");
18   }
19   public static void showMondokoro() throws Exception {
20     System.out.println("飛車さん、角さん。もういいでしょう。");
21     System.out.println("この紋所が目にはいらぬか！");
22     doTogame();     // もう一度、とがめる
23   }
24 }
```

次のルールに従い、このクラスを3つのクラスに分割してください。

① commentパッケージに属するZenhanクラスを作成し、前半に実行される2つのメソッドをそこに移動する。
② commentパッケージに属するKouhanクラスを作成し、後半に実行される2つのメソッドをそこに移動する。
③ デフォルトパッケージに属するMainクラスにはmainメソッドだけを残す。そして、このクラスの先頭ではZenhanクラスだけをインポートする。

なお、2つのメソッド宣言についている「throws Exception」の意味は、現時点では理解しなくて構いません(第15章で解説します)。

練習6-2

練習6-1で分割した各ソースファイルをコンパイルし、完成した3つのクラスファイルを適切なフォルダにコピーしてください。その上で、javaコマンドを実行し、プログラムを正常に動作させてください。

練習6-3

showMondokoro()メソッドを修正し、「この紋所が目にはいらぬか!」の後に3秒間の「待ち時間」を入れます。APIリファレンスでjava.lang.Threadクラスを調べ、プログラムを一時的に止めるメソッドを呼び出すよう修正してください。

練習6-4

Windowsの環境変数CLASSPATHとして、「c:¥work¥ex64」が設定されているとします。このとき、現在のフォルダ(カレントディレクトリ)によらず「java Main」というコマンドで練習6-3のプログラムが動作するには、Main.class、Zenhan.class、Kouhan.classを、どのフォルダに配置すればよいか答えてください。

練習6-5

「java Main」というコマンドを実行すると、練習6-3のプログラムが動作するWindowsコンピュータがあります。また、このコンピュータのZenhan.classは、「c:¥javaapp¥koumon¥comment」というフォルダの中に存在しています。このとき環境変数CLASSPATHとして設定されている内容を答えてください。

練習問題の解答

練習 6-1 の解答

Main.java

```
1  import comment.Zenhan;
2  public class Main {
3    public static void main(String[] args) throws Exception {
4      Zenhan.doWarusa();
5      Zenhan.doTogame();
6      comment.Kouhan.callDeae();
7      comment.Kouhan.showMondokoro();
8    }
9  }
```

前半（4〜5行目）
後半（6〜7行目）

Zenhan.java

```
1  package comment;
2  public class Zenhan {
3    public static void doWarusa() {
4      System.out.println("きなこでござる。食えませんがの。");
5    }
6    public static void doTogame() {
7      System.out.println("この老いぼれの目はごまかせませんぞ。");
8    }
9  }
```

Kouhan.java

```
1  package comment;
2  public class Kouhan {
3    public static void callDeae() {
4      System.out.println
         ("えぇい、こしゃくな。くせ者だ！であえい！");
```

```
5    }
6    public static void showMondokoro() throws Exception {
7      System.out.println("飛車さん、角さん。もういいでしょう。");
8      System.out.println("この紋所が目にはいらぬか！");
9      Zenhan.doTogame();    // もう一度、とがめる
10   }
11 }
```

練習 6-2 の解答

ここでは一般的と考えられる方法を示します。他に、環境変数を設定するなどの方法があります。

1. コンピュータに適当なフォルダ(たとえば、c:¥japp とする)を作成する。
2. C:¥japp フォルダの中に、Main.class をコピーする。
3. C:¥japp の中に comment というフォルダを作成する。
4. C:¥japp¥comment の中に、Zenhan.class と Kouhan.class をコピーする。
5. C:¥japp を現在のフォルダ(カレントディレクトリ)とする。
6. 「java Main」として java コマンドを実行する。

練習 6-3 の解答

showMondokoro( ) メソッドのみを抜粋してあります。

```
1  public static void showMondokoro() throws Exception {
2    System.out.println("飛車さん、角さん。もういいでしょう。");
3    System.out.println("この紋所が目にはいらぬか！");
4    Thread.sleep(3000);    // この行を追加
5    Zenhan.doTogame();     // もう一度、とがめる
6  }
```

練習 6-4 の解答

Main.class → c:¥work¥ex64 フォルダ
Zenhan.class と Kouhan.class → c:¥work¥ex64¥comment フォルダ

練習 6-5 の解答

c:¥javaapp¥koumon

似ているようで異なる java と javac の引数

付録 A.3.4 項で「javac でコンパイルするときにはソースファイル名に拡張子 ".java" を付けるが、java コマンドでクラスファイルを実行するときには拡張子は不要」と強調しました。なぜ、このような違いがあるのでしょうか？

実は、java コマンドと javac コマンドの引数には次のような意味の違いあるのです。

- **javac コマンドは「どのソースファイルをコンパイルするか」をファイル名で指定して実行するもの**
- **java コマンドは「どのクラスの main メソッドを起動するか」をクラス名（FQCN）で指定して実行するもの**

両者は、まったく別のものを指定するコマンドだったのですね。

すっきり納得 オブジェクト指向

Java の本当のおもしろさ

ここまでお疲れさま。どうだい、Java とはいい友だちになれそうかい？

はい！　…いや、実はちょっと迷うこともあるけど…。でも基本的には main メソッドの中に処理を書いていけばいいんですよね？

専門家も使う Java って、超ムズかしくて、完全に意味不明なんじゃないかって想像してたのに、安心したっていうか、ちょっと拍子抜けっていうか…。

ははは、それはよかった。でも、2 人はまだ Java の本当の姿、本当の魅力にほとんど触れていないんだよ？

えっ？

Java はここからが大事だし、おもしろいんだ。少し学習のレベルは上がるけど、Java の本当の魅力に触れれば、2 人のプログラミング観がきっと変わると思う。

第 I 部で学習した文法だけを使うことでも Java のプログラミングは行えます。しかし、Java というプログラミング言語の真価は、「オブジェクト指向」という概念と組み合わせて初めて発揮されます。
第 II 部では、Java をマスターする上で最も重要と言われるオブジェクト指向プログラミングについて、1 つずつていねいに学んでいきます。

第7章

オブジェクト指向をはじめよう

この第7章からはJavaの根幹となる「オブジェクト指向」を
学んでいきます。オブジェクト指向をラクに理解できるかどうかは、
「学び方のコツを知っているか」「準備体操をしているか」によって
変わってきます。
そこで本章では、オブジェクト指向に本格的に取り組む前段階として、
「その全体像と学び方」を多くのイラストを交えてやさしく解説していきます。

CONTENTS

7.1 オブジェクト指向を学ぶ理由

7.1.1 ソフトウェア開発の新たな課題

菅原さん……助けてください！（泣）

どうしたんだい？ 先週までは、『ボクはもうどんな大きなプログラムだって書けるんです！』と自信満々だったじゃないか。

はい、文法はすべてわかりますし、必要な命令も自分で調べられます。「RPG：スッキリ魔王征伐」の開発も順調でしたが、ソースコードが400行を超えたあたりで、何というか……どこに何の処理を書いたのかわからなくなって、自分でも頭が混乱してしまいました。機能を修正しようとするたびに「**頭がパンクしそうになって、開発が進まない**」んです。

『わからないことはないのに書けない』んだね。それじゃ、その悩みを解決するための「**オブジェクト指向**」について学んでいこうか。

第Ⅰ部を通して、私たちはJavaの基本文法をひととおり学習しました。また、APIリファレンスを調べることで、Javaに用意されたさまざまな命令を利用できるようになりました。つまり、一部の特殊な例を除けば、理論上どのように大きなプログラムでも書くことができるようになったはずです。

しかし、実際に本格的なプログラムを開発し始めると、湊くんのように「**ソースコードが長く複雑になりすぎて、開発者自身が把握しきれなくなる**」という課題に直面します。

この課題は、第6章で学習した方法を用いてソースコードを複数のクラスやメソッドに分割することで、多少は緩和されます。ですが、それでもソースコードが数千行、数万行を越えると、結局は同じ問題に悩まされることでしょう。

実は、この課題は40年ほど前に世界中のプログラマたちがぶつかった壁そのものです。1970年代、さまざまなプログラム言語が登場し、これにより大規模なプログラムの記述が理論上は可能となっていました。

ですが、いざ大きなプログラムを記述しようとすると、人間の頭が追いつかず、開発に時間がかかったり、完成しても不具合だらけのプログラムになったりしてしまうことが少なくありませんでした。なぜなら、その原因はコンピュータの演算性能や記憶容量ではなく、図7-1にあるように**人間がプログラム開発のボトルネックになってしまっている**ことにあったからです。

図7-1　人間自身が巨大なプログラムを把握できなくなり「ボトルネック」になってしまった

7.1.2 オブジェクト指向プログラミングをマスターしよう

そこで、誕生したのが**オブジェクト指向プログラミング**（Object Oriented Programing = OOP）という考え方です。この考え方に従ってプログラムを書くと、前述のような課題に悩むことなく、**大規模なプログラムもラクに開発できる**ようになるのです。

第Ⅱ部（第7章〜第13章）では、このオブジェクト指向の考え方を学んでいきます。オブジェクト指向をマスターすることで、仕事などで携わる大規模で複雑なプログラムも、スッキリと開発できるようになるでしょう。

オブジェクト指向の目的

「人間が把握しきれない複雑さ」を克服するためにオブジェクト指向は生まれた。

7.1.3 オブジェクト指向を学ぶコツ

あぁ！もうダメだぁ！ ボク「オブジェクト指向」って聞いたことがあるんです。とにかく難しくて、挫折してしまう人が多いって……。

その半分は正解だが、半分は間違っているよ。確かにオブジェクト指向をマスターできずに挫折する人はいるし、その本質を理解せずに使っている人もいる。ただ、「**オブジェクト指向の難しさは『学び方』によって大きく変わる**」んだ。

ということは、うまく勉強すれば難しさを感じずにマスターできるってことですか？

そう。君たちの先輩には「オブジェクト指向って、思っていたより簡単ですね」と話す人もいたよ。しかも、その人はプログラミングの経験がなくて、文系出身の人だったんだ。

オブジェクト指向を学ぶ人々からは、「学習が難しい」あるいは「挫折してしまった」、「理解できていない部分もあるが、なんとなく使っている」という声を聞くことも少なくありません。しかし安心してください。「Javaの基本文法は理解できたのにオブジェクト指向はとても難しく感じる」という人には共通点があって、彼らは次のような**たった1つの簡単な前提知識を知らずに、いきなりオブジェクト指向の学習を始めてしまっているだけ**なのです。

基本文法とオブジェクト指向とでは、そもそも学ぶものも学び方もまったく違う

本書の第I部で学んだ内容は、「こう書けば、こう動く」「こう書かないと正しく動かない」という文法や記述のルールでした。書き方の「**正解自体を学ぶ**」のですから、単に丸暗記して習ったとおりに使えばよいだけです。算数にたとえれば計算問題（足し算や引き算の方法を知る）、料理でいうならレシピ（カレーの材料と作る手順を知る）のようなものです。

一方、この第II部では、「プログラムを作るときには、このようにプログラム全体を捉えて、こういう文法の組み合わせ方をすると、ラクにプログラムを作れる」という考え方を学びます。つまり**「正解に辿り着くための考え方」を学ぶ**のです。算数ならば文章題（どう問題を捉えてどう計算していくと答えが出るかという考え方を学ぶ）、料理ならもてなし方（どのような状況で、どのような料理を組み合わせて出せば喜ばれるかという考え方を学ぶ）に相当するでしょう。

このように、第II部では「捉え方」「考え方」を学んでいきます（図 7-2）。つまり、**オブジェクト指向の学習においては、理解することやイメージすることを、より大事にする必要がある**のです。

本書の部・学ぶ趣旨	第I部 基本文法 （定義・ルールなど）	第II部 オブジェクト指向 （考え方）
算数にたとえると?	計算問題を学ぶ （四則演算の意味、書き方など）	文章題を学ぶ （文章の内容を、どう捉えるか）
料理にたとえると?	料理のレシピを学ぶ （いつ、何の材料を、どれだけ入れるか）	料理での「もてなし方」を学ぶ （お祝い事を、どのように捉え、料理の食材で表現していけばよいか）

図 7-2　本書の第I部では Java の基本文法、第II部では「オブジェクト指向の考え方」を学ぶ

7章

丸暗記やひたすら練習みたいな「計算問題の学び方」でがんばっても、文章題はなかなか解けるようにならなくて当然ですね。

そう。オブジェクト指向本来の「考え方」を理解する前に、いきなり小難しい文法を頭に詰め込もうとするから挫折しちゃうんだよ。

オブジェクト指向を学ぶにあたり、まず必要なことは、その「考え方」の概要や全体像を理解することです。この第7章はそのためだけにあります。

しかし、オブジェクト指向という「考え方」には形がなく、把握しづらいところがあるのも事実です。これは「出世する人の仕事に対する考え方」や「恋も仕事もうまくいく人の考え方」などと似ていて、学んですぐに隅々まで100%理解できるようなものではありません。

完璧主義の人にはやや気持ちが悪いかもしれませんが、初めは「こういう感じで考えるんだな」という、多少あいまいでぼんやりとした理解で構いません。繰り返し学習したり、たくさんプログラムを組んだりしていくうちに、徐々に明瞭なイメージになっていくでしょう。

むしろ、一部の章だけを切り出して完璧に理解しようとしても、思ったように理解は進みません。なぜなら、私たちが今から学ぼうとしているオブジェクト指向には、さまざまな「発想」「着眼点」「テクニック」そして「関連する文法」が含まれており、それらは相互に、しかも密接に関係しているからです。第Ⅱ部の学習は、図7-3のように少しずつ繰り返し学んでいくのがコツなのです。

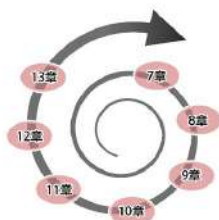



図7-3　Javaの基本文法（第Ⅰ部）は着実に、オブジェクト指向（第Ⅱ部）は繰り返し学んで徐々に理解しよう

7.2 オブジェクト指向の定義と効果

7.2.1 オブジェクト指向の定義

オブジェクト指向を初めて学ぶ人が最初にぶつかる壁があります。それは「オブジェクト指向とは何か？」という問いに明確な答えが得られないことです。

先輩プログラマに質問しても、それぞれ答えが異なり、あいまいで長くてわかりづらい説明をされることもあるでしょう(図 7-4)。正確な答えを知ろうとして、教科書的な定義を持ち出されると、ますます混乱していきます。

図 7-4 「オブジェクト指向とは」という質問が一番難しい?

オブジェクト指向を初めて学ぶ私たちには、小難しい学問的な定義よりも、以下のようなシンプルな定義のほうが理解しやすいでしょう。

オブジェクト指向の定義

オブジェクト指向とは**ソフトウェアを開発するときに用いる部品化の考え方**のこと。

「部品化」という言葉は第６章で学びましたね。１つの巨大なmainメソッドを作る代わりに、複数のメソッドやクラス（ソースファイル）に分割したり、複数の部品を組み合わせたりして、全体として１つのプログラムを作る手法のことでした（図7-5）。

図7-5　巨大な１つのmainメソッドを複数のメソッドやクラスに分割してプログラムを作る

しかし第６章では、「どういう基準で部品を分けるべきか」という**部品化のルールについては触れていません。**プログラムを開発していて、ある程度大きくなって読みづらくなってきたから分割する、なんとなく意味が似ている単位で分割するなど、その程度の主観的な判断にすぎず、「**こう分割したほうが良いプログラムが作れるという確固たる根拠に基づいた部品化**」ではありませんでした。

しかし、これから学習するオブジェクト指向という考え方に沿って１つのソフトウェアを複数に部品化すると、プログラムが把握しやすくなり、**「人間の頭が追いつかない状況」を避けることができる**ようになるのです。

7.2.2 オブジェクト指向のメリット

先輩、私、オブジェクト指向の目的やメリットは本で読んだことがあります。大規模なプログラムを作る際に、「柔軟性が上がる」「再利用性が上がる」「保守性が上がる」と書いてありました。賢い部品化で「複雑なプログラムを人間が把握しやすくなる」から、その結果、柔軟で保守・再利用しやすいプログラムが作れるんですね。

確かにオブジェクト指向の入門書には、そのような説明が書いてあるね。でも、今の朝香さんにとって、その理解でいいのかな？

いいも何も、オブジェクト指向って、そういうものじゃないんですか？

オブジェクト指向とは「何のため」にある考え方なのでしょうか。利用すると「何が嬉しい」のでしょうか。

その根底にある目的は、**「人間に把握できるプログラム開発を実現する」**というものです。この考え方を利用した「賢い部品化」を行うと、把握しやすさの向上のほかにも次のようなメリットが生まれるといわれます。

- **プログラムを容易に変更しやすくなる（柔軟性・保守性の向上）**
- **プログラムの一部を簡単に転用できる（再利用性の向上）**

しかし、Javaを学び始めたばかりのみなさんは、このメリットを**一度忘れてください**。これら柔軟性・保守性・再利用性のメリットは、厳しい予算や納期の中で大規模なプログラムを何度も開発・修正するようになって初めて実感できるものです。

つまり、本書を学習しているみなさんは、先ほど述べた「保守性」や「再利用性」の必要性や大切さを心の底から理解し、**「保守性や再利用性のために、オブジェクト指向を絶対マスターするぞ！」とは思えないはず**です。

オブジェクト指向は「一度マスターしてしまえば、二度と手放せないぐらい便利な一生モノの技術」です。しかし、「鼻歌交じりにナメてかかって理解できる」ほど生やさしいものではありません。これからオブジェクト指向を学ぶみなさんは、今この章で**「マスターできたら嬉しい！」「絶対マスターしたい！」と心底思えるようなメリットをイメージしなければなりません**。

今の私たちが抱くべき「オブジェクト指向のメリット」は、次のひとことで十分です。

オブジェクト指向を用いるメリット
「ラクして、楽しく、良いもの」を作れる

同じ開発をするのなら、毎晩のように徹夜をしたり休日出勤したりして開発するのと、毎日定時に帰れるよう効率的に開発するのと、どちらがよいでしょうか？

頭をかきむしりながら画面とニラメッコして開発するのと、まるで絵でも描くように創造力を発揮しながらプログラムを作っていくのでは、どちらがよいでしょうか？

聞くまでもありませんね。みなさんはこの第Ⅱ部の内容をマスターすることによって「ラクして、楽しく、良いものを」作れるようになるのです。

「考え方」「捉え方」の違いが世界を変えることもある

私たちは日常生活でも「考え方」を変えることによって生活を便利にしています。たとえば「ゼロ」や「マイナス（負数）」の考え方です。

天気予報の気温などでは「○○地方の最低気温はマイナス 10 度」などと表記されます。日常では当然のように見かけるマイナスの数字ですが、原始時代の人類にとって、自然界の何かを数える場合は常に 1 以上でした（そのため 1 以上の整数を「自然数」といいます）。

「なにもないこと（ゼロ）」や「なさすぎること（負数）」を数字として扱う「考え方」を導入したのは、人類の長い歴史の中でもわずか 1400 年ほど前からです。ゼロや負の数という考え方を導入する前と後で世界が変わったわけではありません。**人間が「考え方」「捉え方」「概念」を変えただけ**なのです。

しかし、この新たな「考え方」の導入によって、人間は世界のさまざまなものを数字として把握、制御することが可能になり、IT のみならず社会生活を発展させる基盤となっています。

オブジェクト指向の全体像と本質

7.3.1 オブジェクト指向と現実世界

でも菅原さん、たかが「ある考え方」を利用するだけで、なぜラクして楽しく良いものが作れてしまうんでしょうか？ その「なぜ」がわからないとスッキリしません。

そうだね。それはオブジェクト指向の全体像と、その本質を知れば理解できるよ。

オブジェクト指向という考え方を採用し、部品化をするだけで、「ラクして楽しく良いもの」が実現できてしまうなんて、何だか悪徳商法の宣伝文句のような印象を持たれる方もいるかもしれません。この節では、その根拠となるオブジェクト指向の本質を探っていきましょう。

そもそも、我々は何のためにプログラムを開発するのか、考えたことはあるかな？

普段あまり考えることはないかもしれませんが、私たちが開発するプログラムやシステムは、**「現実世界における何らかの活動を自動化するためのもの」**です。さらにわかりやすく言うなら、「人がやってきたことを機械にやらせて人がラクをするためのもの」と言い換えることもできるでしょう。

たとえば、銀行のATMシステム（ATMの機械と、その中で動いているプログラム）を想像してください（次ページ図7-6）。江戸時代のように、コンピュータがなかった頃には手作業で行っていた作業（依頼の受け付け、残高の検査、引き出し、記帳、受け渡し）を、機械に肩代わりさせていると考えることができます。

現代

①ATMを操作して
5,000円を引き出す

ATM

②データベースに
残高を問い合わせる

残高
データベース

④5,000円を渡す

③5,000円を減らすよう
データを書き換える

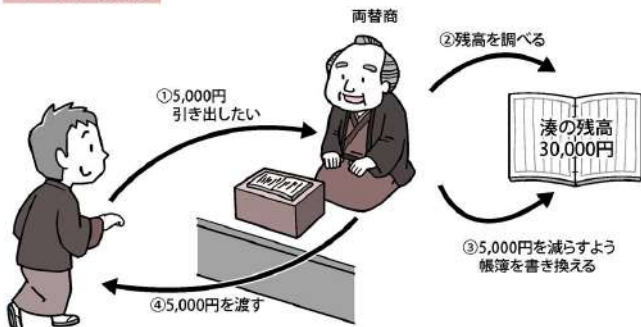
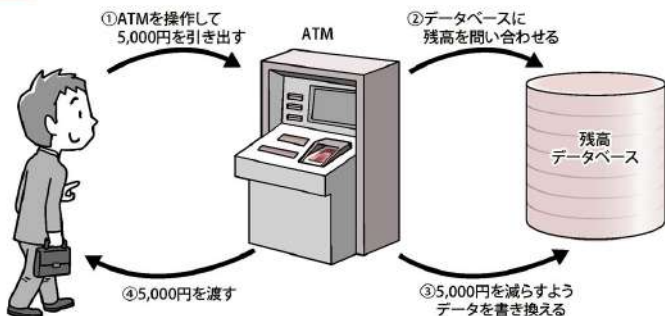



図 7-6　ATM システムは、江戸時代における「両替商」の代わりであり、人間がやっている作業をプログラムに置き換えたもの

同様に、現実世界での「切符販売の駅員の仕事」を機械とプログラムに置き換えたものが、「自動券売機とそのプログラム」です。現実世界での「みんなに公開する日記帳」をプログラムにしたものが「ブログ」です。また、本書の湊くんが作っているRPGも現実世界ではありませんが、ファンタジーの世界のさまざまな人物の冒険や戦いをコンピュータ上で実現しているものです。

このようにプログラムやシステムは、現実世界のある活動を人間に代わって機械にやらせるために作られるものであって、**現実世界とは無関係に単独で存在しているものは、ほとんどありません。**

7.3.2 手続き型プログラミングとの違い

第 I 部で私たちが行ってきた従来のプログラミング手法は、**手続き型プログラミング**（procedural programming）と呼ばれています。プログラマは頭を捻り、コンピュータがどのように動けばよいかという手順を考え、プログラムの先頭から順番に命令として記述していく方法です。

一方、オブジェクト指向で開発を行う場合、プログラマはいきなりコードを書き始めることはしません。まず、プログラムで実現しようとする部分の「現実世界」を観察します。たとえば銀行振込の手続きをプログラム化する際には、それを観察して図 7-7 のようなイメージ図（設計図）を描きます。

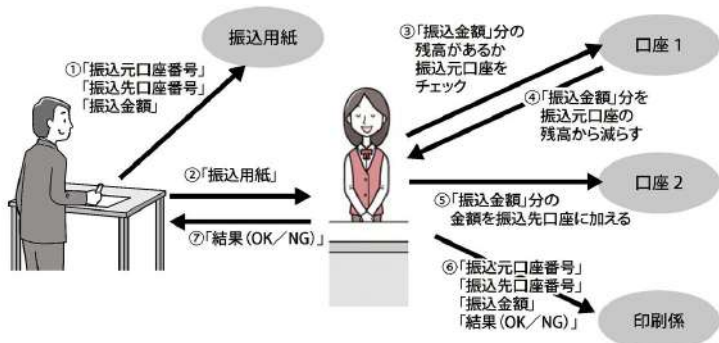



図 7-7　オブジェクト指向では、まず現実世界を観察し、それを設計図に落とし込んでいく

ここで着目してほしいのは、この設計図は IT の知識がない一般の人に見せても理解できる「現実世界における銀行取引の構図そのもの」である点です。オブジェクト指向の開発では、設計図の中の登場人物や物の 1 つひとつを部品と捉え、それを「クラス」という Java における部品で記述していくのです（次ページ図 7-8）。

オブジェクト指向による部品化のルール

現実世界に出てくる登場人物の単位で、プログラムをクラスに分割する。

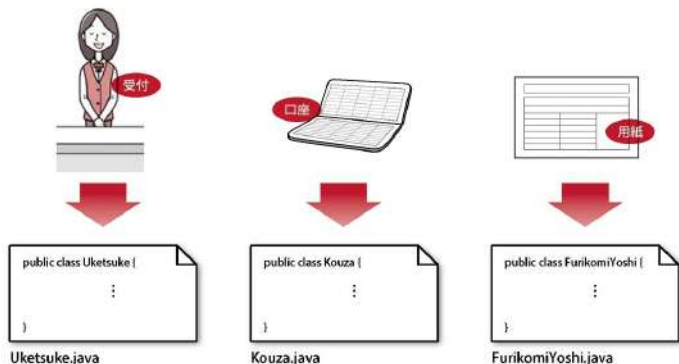



図 7-8　設計図の部品を Java のクラスとして記述していくことで、現実世界の定義や行動をプログラム化できる

7.3.3　開発時に作るクラス、実行時に動くオブジェクト

プログラマが作成する部品（クラス）とは、たとえば ATM のプログラムであれば「Uketsuke.java」のようなプログラムコードです。

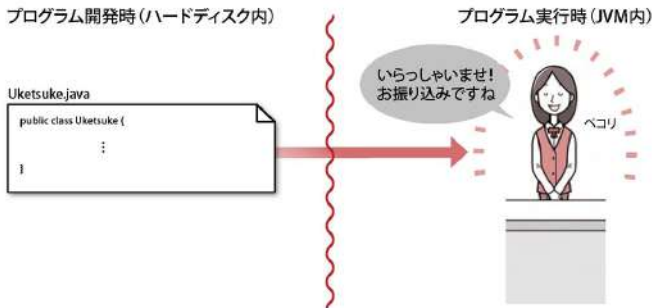



図 7-9　Java で作られた Uketsuke クラスから、JVM 内に「仮想的な受付係」が生み出され、現実世界の動作をまねて動き出す

開発時に作られたクラスは、プログラムとして実行される際、それぞれ「**仮想的な登場人物**」のオブジェクトとして JVM の中にその存在が生み出されます（図 7-9）。

そして図7-10のような「仮想的な口座」「仮想的な受付」「仮想的な印刷担当」などが、コンピュータ(JVM)という「電子的な仮想世界」の中に作られ、現実世界をそっくりまねた「**Java仮想世界**」とでもいえるような世界を形成します。

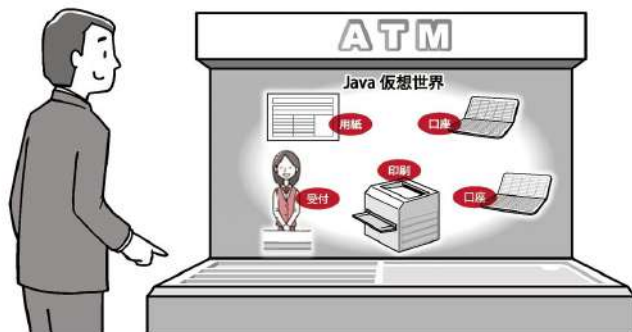



図7-10　ATMは銀行の各種業務をプログラムに置き換えた「電子的な仮想世界」

7.3.4 オブジェクト指向におけるプログラマの役割

オブジェクト指向プログラミングにおいて、プログラマはまるで「Java仮想世界における神様」のような存在です。なぜなら仮想世界にどんな登場人物や物を生み出し、それらをどのように連携させるかを決め、それぞれの部品を作っていく立場だからです。

たとえば国内の複数の倉庫に在庫のある書籍をネットで販売するプログラムを作るなら、本の販売業務を観察した上で、「必要な物(オブジェクト)は『本』『倉庫』『顧客』『購入記録』…」と判断し、図7-7のような設計図を書いていきます。

手続き型のプログラマのように「コンピュータが一行一行、何を実行するかという手順を定める」のではなく、「オブジェクトをどう作るか、どのように連携させるか」を第一に意識しながら開発していきます。このことが「オブジェクト指向プログラミング」という名前の由来になっています。

「〜指向」というのは、「〜を大切にした」「〜を中心に据えた」という意味だよ。

7.3.5 オブジェクト指向の本質

それでは、「なぜオブジェクト指向の考え方を使うと、大規模で複雑なプログラムも把握しやすくなり、その結果、ラクして楽しく良いものを作れるのか？」という問いに答えましょう。

私たち人間が慣れ親しみ、よく把握している現実世界をマネして作られたプログラムもまた、私たち人間にとって把握しやすいものだから

さらにオブジェクト指向には以下のメリットもあります。

- **プログラム開発時に「頭を捻り、発想して作る」必要はありません。現実をお手本に、それをマネして作ればいいのです。**
- **現実世界の登場人物に変化があった場合、対応する部品（クラス）を修正、交換すれば簡単にプログラムを修正できます。**

このようなメリットは、「現実世界をマネる」からこそ生まれてきます。つまり、**現実世界の登場人物たちを、コンピュータの中の仮想世界にオブジェクトとして再現し、現実世界と同じように連携して動くようにプログラムを作ること**こそがオブジェクト指向の本質なのです。

いくらクラスをたくさん使っていても、後述する「継承」などの機能を利用していても、オブジェクト指向の本質を正しく理解しておかなければ、「人間が把握しにくく、修正や交換が困難なプログラム」しか作成できないでしょう。

オブジェクト指向の本質

現実世界の登場人物とそのふるまいを、コンピュータ内の仮想世界で再現する。

7.4 オブジェクトと責務

7.4.1 サッカーで考えるオブジェクト指向

手続き型とオブジェクト指向との違い、そして「オブジェクト」をより深くイメージするために、サッカーの試合を題材に考えてみましょう。

監督であるあなたは、11 名の選手に的確に指示を出して彼らを動かし、相手チームからゴールを奪わなければなりません。これは、「プログラマであるあなたが、コンピュータに的確に指示を出し、目的どおりに動かさなければならない」状況と似ています。

次の図 7-11 は、「手続き型プログラミング」をサッカーにたとえて表現した図です。

図 7-11 「手続き型」の監督は、選手に逐一、指示を出さなければならない

監督であるあなたは、全選手の一挙手一投足に対して、すべての指示を細かく出す必要があります。これでは監督の負担が非常に大きく、体がいくつあっても足りません。作戦変更・選手交代などをしようとしても、監督自身が大混乱してしまうでしょう。

これってまさに、この章の冒頭のボクの状態ですね。

そして次の図 7-12 は、「オブジェクト指向プログラミング」をサッカーで表現した図です。

図 7-12　「オブジェクト指向型」の監督が事前に選手の役割と責任を割り当てれば、試合では選手自身が自分の役割を果たすために行動する

監督であるあなたは、1 人ひとりの選手を部品と考え、**それぞれの責務（役割・責任）**を事前に割り当てたクラスとして作ります。試合が始まったら、監督のすることはほとんどありません。

仮想世界には、それぞれの選手オブジェクトが生み出され、**後は選手オブジェクト自身が自分の役割を果たしながら他のオブジェクトと連携して動いてくれます。**

監督（＝プログラマ）であるあなたは、それぞれの選手（＝クラス・オブジェクト）に、「この状況下で、どう行動すべきか」という責務をあらかじめプログラミングしているため、試合中にそれぞれの選手の一挙手一投足まで指示する必要はなくなるのです。

責務の割り当て

オブジェクト指向プログラミングでは、プログラマはそれぞれの部品に「責務」をプログラムとして書き込む。

7.4.2 オブジェクトの姿

これまでの例に挙げた「サッカー選手」「口座」「受付」など、仮想世界で動くそれぞれのオブジェクトは、すべて何らかの責務を仮想世界の神様たるプログラマから与えられます。たとえば、「サッカー選手」オブジェクトは「ボールを受けたら前に走る」「シュートする」など、あらかじめ設定された役割を果たす**行動責任**を負っています。

同様に、ATMの「受付」オブジェクトにも図7-7(p.281)のような行動責任があります。振り込み依頼を受け付けたら、「口座」オブジェクトが管理する2つの口座間でお金を移動し、その結果を「印刷係」オブジェクトに渡してATM利用控えの印刷を依頼するという一連の流れが受付の行動責任です。

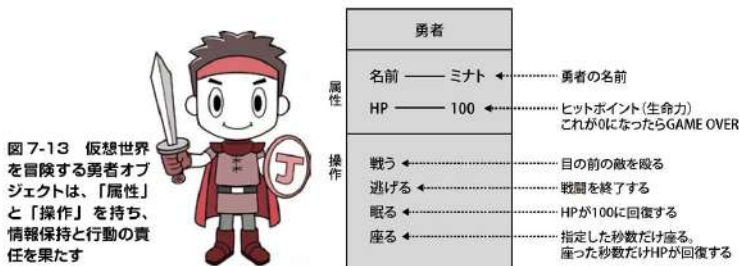
では、「口座」オブジェクトは、いったいどんな責任を負っているのでしょうか？「口座」は行動責任を負っていませんが、「残高をしっかり覚えておく」という**情報保持責任**を負っています。

確かに、気づいたら中身が消えているような「無責任な口座」では困りますね。

このような「情報保持」と「行動」の責任を果たすために、それぞれのオブジェクトは「**属性**」と「**操作**」を持っています。

【属性】　その登場人物に関する情報を覚えておく箱
【操作】　その登場人物が行う行動や動作の手順

たとえば、湊くんが開発中のRPGにおける「勇者」という登場人物は、図7-13

図7-13　仮想世界を冒険する勇者オブジェクトは、「属性」と「操作」を持ち、情報保持と行動の責任を果たす

のようなオブジェクトとして考えることができるでしょう。

勇者オブジェクトは、自分の名前やHPをしっかり覚えておかなければなりません（＝**情報保持責任**）。そして、もし「戦え」と命令されれば勇敢に目の前の敵と戦い、「眠れ」と命令されれば眠って自分のHPを回復させる責任（＝**行動責任**）があるのです。

そのオブジェクトがどんな属性や操作を持つかは、プログラマが部品を作成する際に決定します。そのためには現実世界の登場人物をよく観察し、どのような属性を持ち、どのような操作ができるかを忠実に再現する必要があります。

では、勇者同様に、「お化けキノコ」という登場キャラクター（これもオブジェクトです）を考えてみましょう（図7-14）。

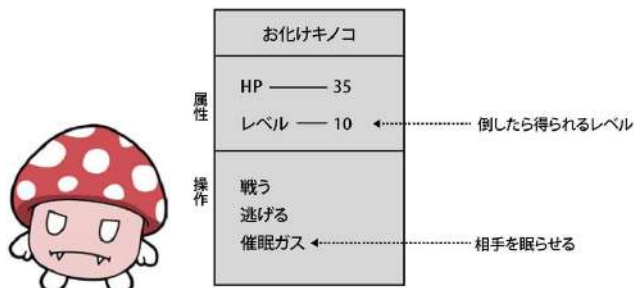



図7-14　仮想世界で「勇者」オブジェクトの敵となる「お化けキノコ」オブジェクトは、勇者と同じく「属性」と「操作」を持っている

お化けキノコは特に重要でないモンスターなので名前は不要と考え、「名前」属性は持っていません。また、このモンスターは「催眠ガス」という技が使えるという設定に従って、その操作を持っています。

7.4.3　オブジェクトのふるまいと相互作用

勇者やお化けキノコは複数の**操作**を持っています。そしてプログラムのmainメソッドや他のオブジェクトから、それらオブジェクトの操作を呼び出す（＝**行動指示を送る**）ことができます。

たとえば、プログラムのmainメソッドから勇者に「座れ」という指示を送れば、勇者は仮想世界内で座って自分のHP属性を回復させる動きをします（図7-15）。なぜなら勇者には、その行動責任があるからです。

同様に、お化けキノコに対して「逃げろ」と指示を送れば、仮想世界内のお化けキノコは逃げ出して戦闘が終わります。

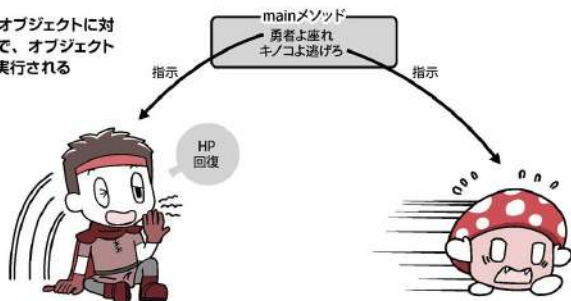
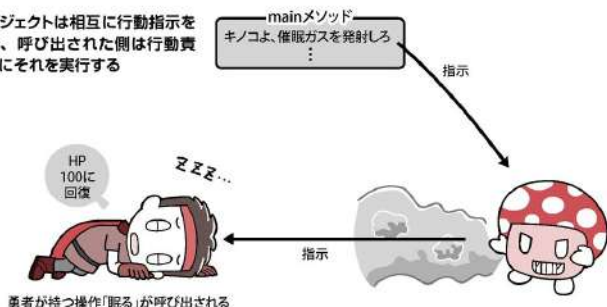
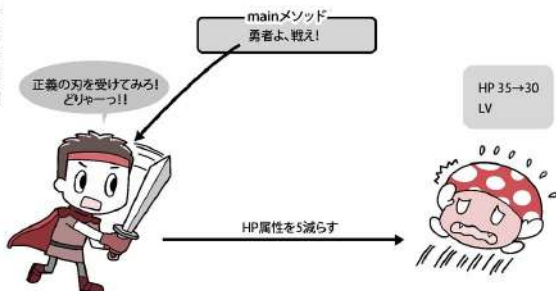
図 7-15　それぞれのオブジェクトに対し行動指示を送ることで、オブジェクトの操作が呼び出されて実行される

また、あるオブジェクトから別のオブジェクトへ操作の指示を送ることも可能です。main メソッドからお化けキノコに「催眠ガス」という指示を送ると、お化けキノコは勇者が持っている操作の中から「眠る」を呼び出します。すると勇者は眠ってしまいます（図 7-16）。

図 7-16　オブジェクトは相互に行動指示を送ることができ、呼び出された側は行動責任を果たすためにそれを実行する

オブジェクトは別のオブジェクトが持つ操作を呼び出すだけでなく、他のオブジェクトの属性を取得したり書き換えたりもできます。

たとえば main メソッドから勇者に「戦う」という指示を送ると、次ページの図 7-17 のように勇者は戦い、結果としてお化けキノコの HP 属性を書き換えて減らす動作をするでしょう。

図 7-17　勇者がお化けキノコを攻撃するということは、勇者がお化けキノコに対してHP属性を書き換えるように行動指示を送る行為だといえる

このように考えると、コンピュータの中の仮想世界で各オブジェクトが互いの属性を書き換えたり操作を呼び合ったりして、物語を繰り広げていく姿が目に浮かびませんか？

このようにして仮想世界内のオブジェクトは互いに「属性」を読み書きしたり、「操作」を呼び出したりして連携し、全体では１つのプログラムとして動きます。

そして、仮想世界の中で「仮想的な受付」や「仮想的な口座」が現実同様に正確に動いてくれるからこそ、現実世界の「本物の受付係」や「紙の口座帳簿」は仕事から解放され、コンピュータによる自動化が可能になるのです。

ここで紹介した勇者とお化けキノコの戦いは、次の第８章でプログラミングして動かしてみるよ。今は「仮想世界の中でオブジェクトたちが互いの属性や操作を呼び出し合って動作する」姿をイメージできれば十分だ。

7.5 オブジェクト指向の3大機能と今後の学習

7.5.1 3大機能とその位置付け

「オブジェクト指向といえば継承」と聞いたことがありますけど、継承は大事じゃないんですか？

さすが朝香くん。もうそんな専門用語を知っているんだね。もちろん、まだ解説していない内容だから知らなくても問題ないけどね。

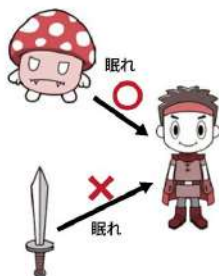
その「継承」って何ですか？

オブジェクト指向の本質に沿った開発をするための手伝いをしてくれる、便利な機能のひとつだよ。

オブジェクト指向の本質は、あくまでも「現実世界を仮想世界内に再現すること」です。よって、現実世界を観察し、「口座」や「受付」という単位でクラスを分割して適切な責務を与えたプログラムであれば、それだけで十分にオブジェクト指向の考え方に沿ったプログラムと言えます。

さらに、Javaのような**オブジェクト指向言語**（Object Oriented Programming Language＝オブジェクト指向の考え方に沿ってプログラムを作りやすい配慮がなされているプログラミング言語）には、開発者が「より便利に」「より安全に」現実世界を模倣できるよう、文法などに専用の機能が準備されています。それが次ページの図7-18に示した**オブジェクト指向の3大機能**です。

カプセル化

属性や操作を、一部の相手からは利用禁止にする機能

現実世界では、剣が勇者に「眠れ」という指示を出すことは、まずありえない

「眠る」操作は、剣オブジェクトから呼べないようにしておいたほうが安全

継承

過去に作った部品を流用し、新しい部品を簡単に作れる機能

すでに「勇者」という部品があれば、空を飛べる「スーパー勇者」は簡単に開発できる

多態性

似ている2つの部品を「同じようなもの」と見なし、「いいかげん」に利用できる機能

「お化けキノコ」と「オオコウモリ」では厳密には斬りつけ方が微妙に異なるはず

違いを気にせず、どちらも「同じようなもの」と見なし、「戦う」操作で攻撃できる

図 7-18　オブジェクト指向の 3 大機能「カプセル化」「継承」「多態性」を利用することにより、便利で安全なプログラムを作ることができる

この 3 大機能については、それぞれ第 10 〜 13 章で詳しく解説していきます。今は「オブジェクト指向の実践を支援する、このような 3 つの機能があるんだな」と、あいまいなイメージで捉えておいてください。

7.6.2　以降の章の学び方

オブジェクト指向の全体を一軒の家になぞらえると、第Ⅱ部の章（7 〜 13 章）は、それぞれ次の図 7-19 のような構造に相当します。

この第 7 章と次の第 8 章がすべての基礎であり、その上に「さまざまなクラス機構」（第 9 章）と、「オブジェクト指向の 3 本柱」（第 10 〜 13 章）が載っています。

本章の冒頭でも紹介しましたが、これから本格的にオブジェクト指向を学ぶにあたっては、次の点に注意してください。

図 7-19　第 II 部の各章で解説するオブジェクト指向の構造

●1章ずつ完璧に理解して次に進む必要はない

第Ⅱ部で学ぶ内容は、それぞれの章が密接に関係しています。そのため「**後の章を学ぶことで、前の章の内容をより深く理解できる**」ということもよくあります。それぞれの章を1度読んだだけでは完全に理解できなかったとしても、とりあえず疑問点は置いて次の章に進んでください。

●最初から、すべての章を理解できなくてもよい

第7章から第13章までのすべてを理解しなければ、オブジェクト指向をまったく使えないわけではありません。この第7章と第8章を理解しておけば、最低限のオブジェクト指向プログラミングは可能になります。第10～13章で紹介する3大機能は、マスターできたものから少しずつ使っていけばよいのです。

とはいえ、カプセル化や継承の基本を理解していないとプログラミングの実務に支障があるため、本書では次ページの図7-20のような学び方をお薦めします。

まずは次ページ図7-20にある1周目の内容までを理解していれば、とりあえずオブジェクト指向を用いた開発ができるでしょう。初めてJavaを学習する人は、まず1周目の部分を理解することを目標にしましょう。

2周目では、「オブジェクト指向の山場」である12～13章に挑みますが、第12章に入る直前に、この第7章をもう一度読み返しておくと、より理解が深ま

ります。

　最後の3周目では、第7章と第8章を復習してオブジェクト指向の全体像と本質を振り返った上で、第12〜13章を読み返すことにより、オブジェクト指向に関する深く実践的な理解を得られるだけでなく、「さまざまなものを自由自在にプログラムで表現できるまでに成長した自分自身の可能性」に気づくでしょう。その段階になれば、きっとオブジェクト指向の楽しさの虜になっているはずです。

本書でのJava学習「周回数」	1周目	2周目	3周目
7章　オブジェクト指向をはじめよう	7〜11	7〜13	7〜8
8章　インスタンスとクラス			
9章　さまざまなクラス機構			
10章　カプセル化			
11章　継承			
12章　高度な継承			12〜13
13章　多態性			
Java習得のレベル	レベル1 要点を押さえ最低限使える	レベル2 ひととおり使うことができる	レベル3 高度に使え活躍できる

図7-20　本書が推奨する学習の流れ（1周目から3周目の学び方）

7.6 第7章のまとめ

この章では、次のようなことを学びました。

学習方法の違い

- 第II部ではオブジェクト指向という「考え方」を学ぶため、「文法」を学んだ第I部とは学び方を変える必要がある。
- 第II部ではイメージを重視し、各章を繰り返し学んでいくことで、ぼんやりとした理解を少しずつ明確にしていく。
- 初めからオブジェクト指向のすべてをマスターする必要はない。まずは第11章までの「1周目」の達成が目標。

オブジェクト指向の概要と本質

- オブジェクト指向とは、ソフトウェアを開発する際に用いる部品化の考え方。
- オブジェクト指向を用いると、大規模で複雑なソフトウェアであっても、ラクして、楽しく、良いものを開発できる。
- オブジェクト指向の本質は、現実世界における「登場人物とそのふるまい」を、仮想世界においても「オブジェクトたちとそのふるまい」として再現すること。
- オブジェクトは属性と操作を持つことによって、現実世界と同様の責務を果たす。
- オブジェクト指向の本質に沿った開発を支援するために準備されている「カプセル化」「継承」「多態性」などの機能を用いたプログラム開発は、第10章〜第13章で学ぶ。

7.7 練習問題

練習問題 7-1

ATM、券売機、ブログなどの他に、「現実世界の人間の活動をプログラムで機械化・自動化しているもの」の例を考えて書き出してみましょう。

（ヒント 1）現代では「機械がやることが当たり前」のことも、昔は人が手作業でやっていたことがほとんどです。

（ヒント 2）ATM や券売機のように「見た目がコンピュータではないもの」の中でもプログラムは動いています。

練習問題 7-2

次のプログラムを作る場合に登場するオブジェクト（現実世界の登場人物）にはどのようなものがあるか、自由に考えて書き出してみましょう。

①現在航行中のすべての飛行機と空港を管理する、航空管制システム

②国内の映画館を選択すると、その映画館での上演映画と、その主演俳優の一覧を表示してくれるプログラム

③余っている食材を入力すると、膨大なレシピの中からその食材を使う料理を検索してくれるプログラム

練習問題 7-3

ある都市の観光案内所には、タッチパネル式の「観光案内端末」が設置されています。利用者が画面から希望条件を入力すると、オススメのお店や名所旧跡の名前・住所・電話番号・解説を提示してくれます。

この観光案内端末の中で動くプログラムの内部では、さまざまなオブジェクトが動作しています。そこで以下２つのオブジェクトが持つであろう「行動責任」「情報保持責任」を自由に考え、操作と属性として書き出してみましょう。

①現実世界の案内係を再現した「案内係」オブジェクト

②現実世界のお店や名所旧跡を再現した「観光地」オブジェクト

（ヒント）オブジェクトには、操作や属性だけを単独で持つものもあります。

練習問題の解答

練習問題 7-1 の解答

この問題に対する解答は無数に考えることができますので、以下に示したのは解答の一例です。

- **電卓の中に入っているプログラムは、「指示したとおりにすばやく計算をしてくれる人」を機械化・自動化したものです。**
- **電子メールは、現実世界の「手紙」を電子化したものであり、そのメールを配送するインターネットのシステムは「郵便配送のしくみ」を機械化したものです。**
- **ネットショッピングサイトは、現実世界の「商店」を電子化したものであり、そのプログラムは従来、店員が受け持っていた「商品の検索依頼・注文依頼・決済」などを自動化したものです。**

練習問題 7-2の解答

この問題の解答も無数に考えることができますので、以下に示したのは解答の例です。

①「飛行機」オブジェクト、「空港」オブジェクト
②「映画館」オブジェクト、「映画」オブジェクト、「俳優」オブジェクト
③「食材」オブジェクト、「レシピ」オブジェクト、「料理」オブジェクト

練習問題 7-3の解答

この問題の解答も無数に考えることができますので、以下に示したのは解答の例です。

①「指定条件に基づいて観光地を検索する」操作
②「名所の名前」属性、「名所の住所」属性、「名所の電話番号」属性、「名所の解説」属性

第8章

インスタンスとクラス

前章では、オブジェクト指向の本質は
「現実世界の登場人物とそのふるまいを
コンピュータ内部の仮想世界で再現すること」にあると学びました。
そこで第8章では、実際にJavaでプログラムを記述し、
仮想世界にさまざまな登場人物を生み出し、
活動させていくために必要な事柄を学んでいきます。

CONTENTS

8.1 仮想世界の作り方

8.1.1 オブジェクトを生み出す手順

第7章で学んだように、オブジェクト指向プログラミングで開発されたプログラムは、動作時に「現実世界をマネた、それぞれのオブジェクトが互いに連携して動く仮想世界」を形成します。そのように考えるとプログラマの仕事は以下の2つといえます。

①各オブジェクトが負うべき責務を考え、「属性」「操作」の種類と内容を定義する。
②各オブジェクトを仮想世界に生み出し、動かす。

手順としては、①オブジェクトを定義して、②オブジェクトを生成すればいいのね。

う〜ん、惜しいね。厳密に言うとJavaでは**「オブジェクトは定義できない」**んだ。

Javaでは、仮想世界の中で動く**オブジェクトそのものをプログラマが直接定義することは許されません**。その代わりにプログラマは、「オブジェクトが生み出される際に用いられる、オブジェクトの設計図」である**クラス**を定義できます。

先ほどの朝香さんの言葉を訂正すると、**①クラスを定義して、②そのクラスに基づいてオブジェクトを生成する**、が正しいのです（図8-1）。

図8-1　クラスの定義とオブジェクトの生成

なるほど。確か7章でも「開発するのはクラスで、動かすときにオブジェクトになる」って学んだわね（7.3.3項を参照）。

8.1.2 クラスとオブジェクトが別である理由

でも先輩、私やっぱり納得がいきません。オブジェクトを直接定義できたほうがシンプルでいいじゃないですか？

確かにそのほうがシンプルだね。事実、オブジェクトを直接定義・生成できる言語も存在する。だけど「クラス定義→オブジェクト生成」という方式にもメリットはあるんだ。

なぜJavaでは「オブジェクト同士が連携する仮想世界」を作るために、わざわざ「クラスを定義して、そのクラスからオブジェクトを生成する」という複雑な手順を踏まなければならないのでしょうか。その答えに辿り着くために、私たちは「**オブジェクトを大量に作る必要がある状況**」を想像しなければなりません。

たとえば銀行で「すべての口座情報の複雑な統計を計算するプログラム」が動くとき、その仮想世界には何個の「口座」オブジェクトが必要でしょうか？ 口座が1,000あるなら1,000個のオブジェクトを生み出す必要があるかもしれません。その**1,000個の口座オブジェクトそれぞれに対して、「属性として、残高・名義人・開設日があって…」という定義を繰り返す必要がある**としたら、それを作るプログラマの作業は膨大なものになります。

そこでクラスの登場です。「属性として、残高・名義人・開設日があって…」という定義をした「口座クラス」を1度作っておけば、この**クラスから100個でも1,000個でも必要な数だけオブジェクトを生み出すことができる**のです。たとえば振り込みを受け付ける「Uketsukeクラス」を1つ準備しておけば、複数の受付係を生み出して、「並行して振り込み依頼を受け付ける」プログラムを作ることもできます（次ページの図8-2）。

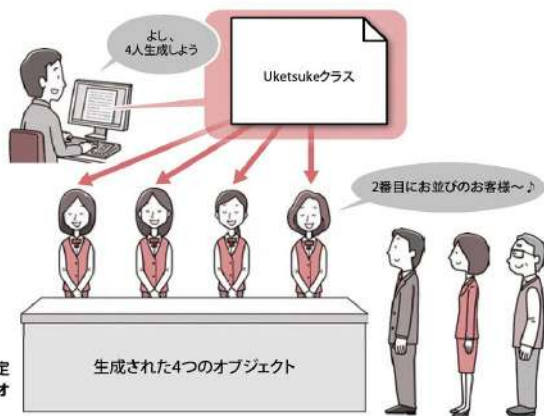



図 8-2　1つのクラスを定義しておけば、何個でもオブジェクトを生成できる

クラスって、プラモデル工場にある「金型」と同じと考えればいいですか？ 元となる金型を1個作っておいて、そこにプラスチックを流し込んで、同じプラモデルを大量に製造するんです。

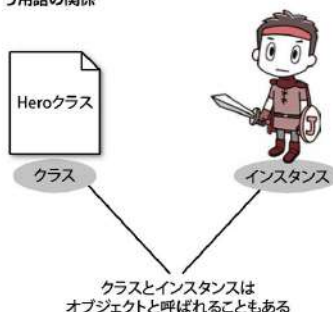
ここで改めて意識してほしいのは、**クラスとオブジェクトはまったく違うものである**という点です。プログラムの動作時に**仮想世界の中で活躍するのは「オブジェクトだけ」であって、その金型であるクラスが仮想世界で活動することは基本的にありません**。「受付係の金型」が挨拶をしたり振り込みを受け付けることはないのです。

8.1.3　オブジェクトという用語のあいまいさ

実は、実際の開発現場における技術的な会話や文章の中で単に「オブジェクト」という表現が用いられる場合、「金型か、その金型から生まれた実体かを厳密に区別しない（会話の都合上、どちらでもいい）」ことがあります。つまり、**「オブジェ**

クト」という用語は、ときどきクラスのことを指して使われることもある、かなりあいまいなものなのです。

もし、「金型ではなく、その型から生み出された仮想世界で活動する実体」を厳密に示したい場合は、**インスタンス**（instance）という用語を用います。また、クラスからインスタンスを生成する行為を**インスタンス化**（instanciation）と表現することもあります。本書でも以降、オブジェクトのことは極力インスタンスと表現します（図 8-3）。

図 8-3　クラス・インスタンス・オブジェクトという用語の関係

インスタンスとクラスの関係

仮想世界で活動するのは「**インスタンス**」であり、そのインスタンスを生み出すための金型が「**クラス**」である。

8.1.4　プログラムに登場する 2 種類のクラス

それではいよいよ、第 7 章で登場した「勇者とお化けキノコの戦い」プログラムを作っていこう。

やったー！ 勇者クラスと、お化けキノコクラスを作るんですよね？

いや、実はもう 1 つクラスを作る必要があるんだ。

第7章で紹介した「勇者とお化けキノコの戦い」をプログラムにするにあたって、まず必要なものとして、勇者(Hero)クラスとお化けキノコ(Matango)クラスはすぐに思いつくでしょう(これらのクラスは「現実世界の登場人物に対する金型」ですので、以後「登場人物のクラス」と表現します)。

図 8-4　勇者やお化けキノコに指示を送る人が誰もいない

しかし、このような「登場人物のクラス」だけではプログラムは動きません。クラスから生み出された勇者やお化けキノコのインスタンスは、**誰かから指示(操作の呼び出し)をもらうことで責任を果たすために動く**ので、この2人だけを開発しインスタンス化しても、いわゆる「指示待ち状態」になってしまうのです(図 8-4)。

図 8-5　勇者やお化けキノコに指示を送る main メソッド

そこで勇者とお化けキノコを作った私たち「神様」が、この2人の登場人物にどのように動くかを指示していく必要があります。私たち神様の指示である「天の声」は、main メソッドとして記述します(図 8-5)。

ただ単に main メソッドを記述したいだけだとしても、メソッドはクラスの中に作るという Java のルールを守る必要があります。よって、Main などの適当な名前でクラスを1つ作り、その中に main メソッドを作ることになります。

リスト 8-1　main メソッドを作る

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      // （以下の内容をJavaで記述していく）
4      // 勇者よ、この仮想世界に生まれよ！
5      // お化けキノコよ、この仮想世界に生まれよ！
6      // 勇者よ、戦え！
7      // お化けキノコよ、逃げろ！
8    }
9  }
```

そして、以下のようにして Main クラスの main メソッドを起動します。

```
>java Main↵
```

これで、勇者やお化けキノコが仮想世界で動き出します。

原理的には、勇者クラスやお化けキノコクラスのどちらかのクラスの中に main メソッドを同居させることもできる。しかし、コードがわかりにくくなってしまうため、Main クラスのように独立したクラスを準備して、その中に作ることをお薦めするよ。

このMain クラスだけは「現実世界の登場人物を模したものではない」ですし、インスタンス化して利用するものでもありません。あくまでも仮想世界の神様として、それぞれの登場人物を生み出し、それらに対して指示を出すことが役割です（本書では、このようなクラスを「神様のクラス」と表現します）。プログラムを作る場合には、「登場人物のクラス」と「神様のクラス」の２種類を作る必要があることを意識しておきましょう。

Java プログラムの組成に必要なクラスたち

・main メソッドを含む、1つの「神様のクラス」
・現実世界の登場人物を模した、複数の「登場人物のクラス」

たとえば「勇者とお化けキノコの戦い」では、次の図 8-6 のように最低でも 3 つのクラスの開発が必要になるでしょう。

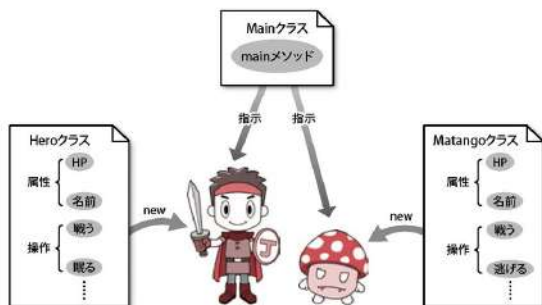



図 8-6　プログラム「勇者とお化けキノコの戦い」全体の構図

以降の解説では、その進行に従って扱うクラスが変わっていきます。その際には「登場人物クラス」の解説なのか、「神様クラス」の解説なのかを意識しながら読み進めるのがスムーズな理解のポイントです。

8.2 クラスの定義方法

8.2.1 登場人物クラスの作り方

それでは、HeroやMatangoクラスを作るために、「クラスの定義方法」を学んでいきましょう。前節で学んだように、クラスには「どのような属性や操作を持っているか」を記述していきます。現時点で私たちが思い描いている「勇者クラスが持つべき属性や操作」は、右の図8-7のようなものです。

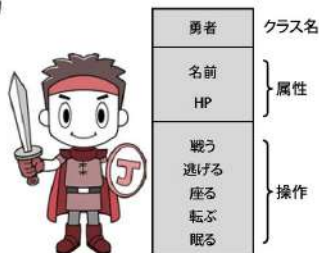



図8-7　勇者クラスの基本設計

なお、この図のように、あるクラスの設計内容を上から「クラス名」「属性」「操作」の一覧として並べる書き方は、**クラス図**（class diagram）という設計図のルールに準じたものです。クラス図は世界共通の設計図として定義された**UML**（Unified Modeling Language）で定められている図の1つです。この基本設計に基づき、実際にJavaでプログラムを書いていくと以下のようなコードになります。

リスト8-2　Heroクラスを Javaのコードで表したもの

```
1  public class Hero {
2    String name;
3    int hp;
4    void attack() {…}
5    void run() {…}
6    void sit(int sec) {…}
7    void slip() {…}
8    void sleep() {…}
9  }
```

ここからは、「クラス名」「属性」「操作」が、どのようにしてコードに書き換わったか、順を追って説明しましょう。

8.2.2 クラスの宣言方法

クラスの宣言文法自体は、第Ⅰ部で学習したものと変わりません。class キーワードを使います。次のリスト 8-3 は中身のない空の Hero クラスを作ってみたものです。

リスト 8-3　中身のない空の Hero クラスを作成

Hero.java

```
1 public class Hero {
2 }
```

第 1 章で学習した「Java のファイル名の約束事」（1.2.1 項）に従い、この場合のファイル名は Hero.java にする必要があることを忘れないようにね。

8.2.3 属性の宣言方法

図 8-7 に示したクラス図によれば、勇者は「名前」と「HP」の 2 つの属性を持っています。そこで、これらの属性について、プログラムで使用する変数の名前と型を考えます。名前は文字列ですから String 型ですね。同様に HP は数値ですので int 型になります。

【名前】　name（String 型）
【HP】　hp（int 型）

これらの変数を、リスト 8-3 の Hero クラスのブロック内に宣言したものが次のコードです。

リスト 8-4　Hero クラスに名前と HP を変数として宣言

Hero.java

```
1  public class Hero {
2    String name;  // 名前の宣言
3    int hp;       // HPの宣言
4  }
```

宣言を追加

上記のようにクラスブロック内に宣言された変数を、Java では特に**フィールド**(field) といいます。これで、name と hp という 2 つのフィールドの宣言が完了したわけです。

フィールドの宣言

属性を宣言するにはクラスブロックの中に変数宣言を記述する。

8.2.4 属性の初期値指定と定数フィールド

ところで、次のようにフィールド宣言と同時に値の代入も行うよう記述しておくと、そのフィールドの初期値を設定することができます（1.3.4 項を参照）。

リスト 8-5　フィールドを宣言すると同時に初期値も設定

Matango.java

```
1  public class Matango {
2    int hp;
3    int level = 10;
4  }
```

お化けキノコのレベルに初期値 10 を設定した

さらに、フィールド宣言の先頭に final を付けると、値を書き換えることのできない**定数フィールド**になります（1.3.5 項を参照）。なお、定数フィールドの名前は一目でそれとわかるように大文字で記述することが推奨されます。

リスト 8-6　フィールドを定数として宣言

Matango.java

```
1  public class Matango {
2    int hp;
3    final int LEVEL = 10;
4  }
```

フィールド LEVEL は 10 で固定

8.2.5 操作の宣言方法

では次に、Hero クラスに「操作」をプログラミングしていきましょう。とはいえ、クラス図のすべての操作を一度に作ることは大変なので、まずは「眠る」操作だけを作ります。「操作」をプログラミングするには、まず「操作の名前」「操作するときに必要な情報の一覧」「操作の結果として指示元に返す情報」「処理内容」の 4 つを考える必要があります。具体的な要素としては以下のようになります。

【名前】　sleep
【必要情報】　なし
【結果】　なし
【処理内容】「眠った後は HP が 100 に回復する」

これらを Hero クラスのクラスブロック内に記述すると次のようになります。

リスト 8-7　「眠る」操作に含まれる要素を記述

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4
5    void sleep() {
6      this.hp = 100;
7      System.out.println(this.name + "は、眠って回復した！");
8    }
9  }
```

自分自身の hp フィールド
自分自身の name フィールド
この部分を追加

これ、第１部でも使っていた「メソッド」じゃないですか？

そうだよ。オブジェクト指向では、ある登場人物の操作を定義するためにメソッドを使うんだ。

メソッド内部の「this.hp」や「this.name」は初めて見る記述ですね。this とは特別に準備された変数で、「自分自身のインスタンス」を意味しています。また、ドット(.)には、日本語でいう「の」と同じ意味がありますので（6.1.2 項）、「this.hp = 100;」は、「自分自身のインスタンスの hp フィールドに 100 を代入」という意味になります。これで、Hero クラスに sleep() メソッドを宣言することができました。

this は省略しないで！

同じクラス内のフィールドへのアクセスの場合、「this.」を省略しても動作します。たとえば「this.hp = 100;」と「hp = 100;」は同じ動作です。しかし、ローカル変数や引数にも同じ hp という名前の変数がある場合、そちらが優先されてしまうなど予想外の動作になる可能性があります。フィールドを用いるときには明示的に this を付けるようにしましょう。

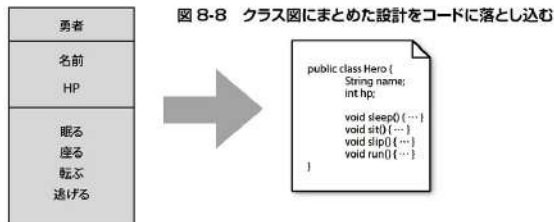
8.2.6 クラス名とメンバ名のルール

クラス名、フィールド名、メソッド名はそれぞれ、基本的に識別子のルール（1.3.2 項）を満たしていれば自由に決めることができます。しかし、実際には次のような慣例に従って名前を付けることが望ましいとされます。

クラス名	名詞	単語の頭が大文字	Hero、MonsterInfo
フィールド名	名詞	最初以外の単語の頭が大文字	level、itemList
メソッド名	動詞	最初以外の単語の頭が大文字	attack、findWeakPoint

8.2.7 クラス定義のまとめ

ここまで、Hero クラスを題材にクラスの定義方法を学びました。クラスを定義するには、まず「属性」「操作」の一覧をクラス図にまとめ、それを Java のコードに置き換えて「フィールド」と「メソッド」として記述していきます（図 8-8）。

図 8-8　クラス図にまとめた設計をコードに落とし込む

この考え方に沿って、勇者クラスに「座る」「逃げる」「転ぶ」メソッドを追加したものが次のコードです。

リスト 8-8　メソッド「座る」「逃げる」「転ぶ」を追加

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    void sleep() {
5      this.hp = 100;
6      System.out.println(this.name + "は、眠って回復した！");
7    }
8    void sit(int sec) {
9      this.hp += sec;
10     System.out.println
           (this.name + "は、" + sec + "秒座った！");
11     System.out.println("HPが" + sec + "ポイント回復した.");
12   }
13   void slip() {
```

sleep メソッド

何秒座るか引数で受け取る

座る秒数だけ HP を増やす

slip メソッド

```
14        this.hp -= 5;
15        System.out.println(this.name + "は、転んだ！");
16        System.out.println("5のダメージ！");
17      }
18      void run() {                          逃げる（run メソッド）
19        System.out.println(this.name + "は、逃げ出した！");
20        System.out.println("GAMEOVER");
21        System.out.println("最終HPは" + this.hp + "でした");
22      }
23    }
```

クラス定義による効果

8.3.1 クラス定義によって可能になる２つのこと

前節で私たちは、Heroクラスを定義することができました。このように、あるクラスを定義することによって、Java言語では２つのことが可能になります。

１つ目は「**そのクラスに基づいて、インスタンスを生成できるようになる**」ことです。私たちは、そもそもインスタンスを生成するために、わざわざクラスを作ってきたわけですから、当然のことですね。

２つ目は「**そのクラスから生まれたインスタンスを入れる変数の型が利用できるようになる**」ことです。たとえば、Heroクラスを定義すると**Hero型の変数が利用できるようになります**。このように、クラスを定義することで利用可能になる型のことを**クラス型**（class type）といいます。

Hero型？ 変数って、数字とか文字列とかを入れるためのものですよね？

そもそもインスタンスを変数に入れることなんてあるんですか？

初めて学ぶことだから、多少の混乱も無理はないね。ではちょっと腰を据えて解説しよう。

8.3.2 クラス型変数とは

まずは湊くんの疑問に答えよう。そもそもHero型の変数ってどんな変数なのか、だね。

Javaで扱うすべての変数は必ず何らかの**型**(type)を持っています。今まで利用してきた「整数を入れるためのint型」や、「文字列を入れるためのString型」は、**Javaが標準で準備しており、いつでも使える型**でした。それに加え、たとえばHeroクラスを定義することで「Heroクラスから仮想世界に生み出されたインスタンスを入れることができるHero型」が使えるようになります。つまり、**クラスを定義すればJavaで利用可能な型の種類はどんどん増えていく**のです(図8-9)。

図8-9　Javaで利用できるクラスたち。新たなクラスを定義することで、その型も利用できるようになる

クラス型変数を準備する方法は、int型やString型と同じです。

リスト8-9

```
1  Hero h;
```

このHero型の変数hには、まだ勇者インスタンスは入っていません。実際にはhに「仮想世界に生み出した勇者のインスタンス」を代入して利用していきます。**インスタンスは通常、クラス型変数に入れて利用する**のです(図8-10)。

図8-10　変数に入れて利用されるインスタンス

8.3.3 クラス型変数が必要な理由

そもそも、なぜHero型のような型にインスタンスを代入する必要があるのでしょうか？ 次のようなケースを想像すれば、「なぜJavaにはインスタンスを入れるための変数の型（＝クラス型）が必要で、クラスを定義することにより利用できるようになるのか」を理解できるでしょう。

図8-11 2つの同名インスタンスを、どのように識別するのか？

図8-11を見てください。これは定義済みのクラスHeroから2つの勇者インスタンスを生み出したものですが、どちらも同じ「ミナト」という名前で、HPも同じく100です。では、この図の右側にいる勇者ミナトに「眠れ」という指示を送るには、どのようにプログラムを記述すればよいでしょうか？

紙面では「右のミナト」「左のミナト」と表現することができますが、**仮想世界の中では2人の勇者を識別する方法がない**ため、指示を送ろうにも相手を特定することができません。しかし、もし2人の勇者が、それぞれ変数h1とh2に入っているとしたら問題は解決します。「h2の勇者に眠れという指示を送る」とプログラムを書けばよいのです。

クラス型変数を用いる理由

仮想世界内に複数存在しうる同名インスタンスの中から、特定の1つのインスタンスをプログラム的に識別するため。

8.4 インスタンスの利用方法

8.4.1 「神様のクラス」の作り方

Hero クラスがやっと完成しました。でも Main クラスを作らないと。

Hero クラスだけではプログラムは動きません。勇者に指示を出す「天の声」が必要だからです(8.1.4 項を参照)。そこで、この節からは、「神様のクラス」である Main クラス(= main メソッド)を作っていきます。Main クラスの大まかな形は次のようなものになります。

リスト 8-10 「神様のクラス」を作る

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3   ここにプログラムの中身を書いていく
4   }
5 }
```

それでは、main メソッドの中に、以下のような内容をプログラミングしていきましょう。

1. 仮想世界に勇者を生み出す
2. 生み出された勇者に、最初の HP と名前を設定する
3. 勇者に「5 秒座れ」「転べ」「25 秒座れ」「逃げろ」と指示を出す

8.4.2 インスタンスの生成方法

では、仮想世界に勇者を生み出すために、「インスタンスの生成方法」について学びましょう。通常、Hero クラスという金型から実体のあるインスタンスを生成するには次のように記述します。

インスタンスの生成

クラス名 変数名 = new クラス名 ();

この文法に従って作成した Main クラスは次のようになります。

リスト 8-11

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     // 1.勇者を生成
4     Hero h = new Hero();
5   }
6 }
```

Hero クラスからインスタンスを生成し、変数 h に入れる

ここで実際にインスタンスを生成しているのは、右辺の「new Hero()」という部分であり、「=」によって生成したインスタンスを Hero 型変数 h に代入しています（この過程の詳細は、次章の 9.1 節で説明します）。

なお、上記の例は 1 行で書いていますが、「Hero h;」「h = new Hero();」の 2 行に分けて記述しても、ほぼ同じ意味です。

これで図 8-10（p.315）の状態になったわけね。

8.4.3 インスタンスのフィールド利用

生み出されたばかりの勇者hには、HPも名前もまだありませんので、それぞれのフィールドに値を代入してあげましょう。勇者hのフィールドに値を代入するには次のような文法を使います。

フィールドへの値の代入

変数名.フィールド名＝値;

フィールド値を取得するには、単に「変数名.フィールド名」と記述します。ここまでの内容を用いてmainメソッドを改良したものが次のコードです。

リスト 8-12

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      // 1.勇者を生成
4      Hero h = new Hero();
5      // 2.フィールドに初期値をセット
6      h.name = "ミナト";
7      h.hp = 100;
8      System.out.println("勇者" + h.name + "を生み出しました！");
9    }
10 }
```

追加した部分（6〜8行目）
変数hのnameに代入（6行目）
変数hのhpに代入（7行目）
変数hのnameを取り出す（8行目 h.name）

8.4.4 インスタンスのメソッド呼び出し

それでは、いよいよ勇者の冒険の始まりです。さまざまな指示を勇者に送りましょう。この「指示」は、実際には「メソッド呼び出し」という形で実現します。

リスト 8-13

Main.java

```
public class Main {
  public static void main(String[] args) {
    // 1.勇者を生成
    Hero h = new Hero();
    // 2.フィールドに初期値をセット
    h.name = "ミナト";
    h.hp = 100;
    System.out.println("勇者" + h.name + "を生み出しました！");
    // 3.勇者のメソッドを呼び出してゆく
    h.sit(5);      // 5 秒座れ
    h.slip();      // 転べ
    h.sit(25);     // 25 秒座れ
    h.run();       // 逃げろ
  }
}
```

では、このコードを実行してみましょう。

```
>javac Main.java  Hero.java⏎
>java Main⏎
```

実行結果

```
勇者ミナトを生み出しました！
ミナトは、5秒座った！
HPが5ポイント回復した.
ミナトは、転んだ！
5のダメージ！
ミナトは、25秒座った！
HPが25ポイント回復した.
```

```
ミナトは、逃げ出した！
GAMEOVER
最終HPは125でした
```

おお！ゲームらしくなってきましたね！

お化けキノコも生み出せば図 8-5（p.304）のようなこともできるよ。

ここでぜひ着目してもらいたいのは、この**mainメソッドの内容が「まるで冒険物語のシナリオ台本みたい」でわかりやすい**点です。もしオブジェクト指向を学習する前の私たちであれば、この画面出力結果を得るために、mainメソッドの中にいくつものHP計算処理やSystem.out.println()呼び出しを繰り返し並べていたことでしょう。たとえば次のようなプログラムです。

リスト 8-14　オブジェクト指向でない方法で作ったプログラム

Main.java

```
1  public class Main {
2    public static void main(String[] args){
3      int yusha_hp = 100;
4      int matango1_hp = 50;
5      int matango2_hp = 48;
6      String yusha_name = "ミナト";
7      int matango1_level = 10;
8      int matango2_level = 10;
9      System.out.println(yusha_name + "は5秒座った!");
10     yusha_hp += 5;
11     System.out.println("HPが5ポイント回復した!");
12     ⋮
```

```
13        ⋮
14    }
15 }
```

リスト 8-13 をもう一度眺めてください。次のような「細かい処理」はいっさい出てきません。

HPを増やしたり減らしたりという細かい計算処理
戦闘中画面にどのようなメッセージを出すかという細かい内容

main メソッドに記述されていることは、ただ単に「勇者を登場させ、座らせ、転ばせ、座らせ、逃げさせる」ということだけです。にもかかわらず、最終的に動いた結果としては、何か行動するたびに HP の増減が処理され、画面には 10 行もの表示がなされ、最終的に HP が 125 であることが正しく出力できています。

小規模な例ではありますが、「オブジェクト指向を使うとプログラムが把握しやすくなる」ことを体験できたでしょうか。

7.4.1 項で出てきたサッカーの例で、監督がそれぞれの選手の一挙手一投足を指示しなくてよかったのと同じね。

うーん、ボクはオブジェクト指向を使わないほうがわかりやすく感じるかもなぁ…。Hero クラスを作ったり、Main クラスを作ったり、あっちこっちのソースコードを見ないといけなくて…。

それは湊くんが「複数のクラスを作っていくこと」に慣れていないからだよ。大丈夫、すぐに慣れるよ。

8.4.5 インスタンス利用のまとめ

ここまでの学習で、定義済みの Hero クラスから勇者インスタンスを生成し、そのフィールドに値を代入したり、さまざまな指示を出したりできるようになり

ました。整理すると以下のようになります。

- **インスタンスの生成には new を使う。**
- **フィールドを利用する場合は「変数名 . フィールド名」と記述する。**
- **メソッドを呼び出す場合は「変数名 . メソッド名 ()」と記述する。**

ここまで学んできたことを使って、仮想世界に勇者とお化けキノコ 2 匹を生み出すプログラムを書くと次のようなものになります。

リスト 8-15　お化けキノコクラスの定義

Matango.java

```
1 public class Matango {
2   int hp;
3   final int LEVEL = 10;
4   char suffix;
5   void run() {
6     System.out.println
          ("お化けキノコ" + this.suffix + "は逃げ出した！");
7   }
8 }
```

リスト 8-16 仮想世界に勇者とお化けキノコ 2 匹を生み出すプログラム

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h = new Hero();
4     h.name = "ミナト";
5     h.hp = 100;
6
7     Matango m1 = new Matango();
8     m1.hp = 50;
9     m1.suffix = 'A';
10
```

勇者を生成し初期化

お化けキノコ A（1 匹目）を生成し初期化

8章

```
11        Matango m2 = new Matango();
12        m2.hp = 48;
13        m2.suffix = 'B';
14
15        // 冒険のはじまり
16        h.slip();
17        m1.run();
18        m2.run();
19        h.run();
20    }
21 }
```

お化けキノコB（2匹目）を生成し初期化

勇者は転ぶ

お化けキノコAが逃げる

お化けキノコBも逃げる

勇者も逃げる

8.4.6 オブジェクト指向のクラスは現実世界とつながっている

クラスの作り方・使い方は理解できました。でも、なんだか不思議な感覚だなぁ…。クラスやメソッドって第Ⅰ部でも使ってきましたよね？

私も。今までも多くのクラスやメソッドを作ってきたのに、この章で作ったクラスは印象がまったく違っていて、とても同じものとは思えないわ…。

この第8章でHeroクラスのコードを学んだ後に、第Ⅰ部（第1～6章）のコードと見比べてください。**それぞれの印象が大きく違う**と思いませんか？ 第Ⅰ部のクラス（HelloWorldなど）と、第Ⅱ部のクラス（Heroなど）は、どちらも同じ「public class クラス名」で宣言されたクラスで「**文法的には両者とも正しいクラス**」です。しかし、この両者には「クラスを何のために用いるか」という**思想が伴っているか否かに決定的な違い**があります。

次ページの図8-12の右側にあるように、第Ⅰ部で学んだHelloWorldなどのクラスには、何を基準にクラスとするか、何を基準にメソッドとするかという明確な思想はありませんでした。強いて挙げるなら「そろそろメソッドが大きくなっ

てきたので分割しよう」、あるいは「この機能のメソッドはこのクラスにまとめておこう」などといった、**開発者の都合や機能の単位などでクラスやメソッドを作ってきた**と言えます。

一方、図 8-12 の左側にある第 8 章の Hero クラスなどは、**「オブジェクト指向」という明確な思想に基づいてクラスやメソッドが作られています**。すなわち現実世界の登場人物を 1 つのクラスとして捉え、その登場人物の持つ属性をフィールドに、そして操作をメソッドとしてコードを書いたのです。

第 I 部では「現実とは無関係のクラス」のみを取り扱ってきましたが、読者のみなさんは、この第 8 章で初めて**「現実世界と意味がつながったクラス」**と出会いました。それが「印象が違う」と感じた理由です。

オブジェクト指向を意識したプログラム開発とは、**「現実世界の人や物、出来事をクラスに置き換えていく」**作業にほかなりません。たとえば、自動車組立て工場のシステムを作るなら「Car」や「Engine」といった部品のクラスを作ります。あるいは学生と教員を管理するプログラムなら、「Student」や「Teacher」といったクラスを作ることになるでしょう。

現実に似せて作り、現実に似せて動かしていく、これがオブジェクト指向の根本的な思想なのです。

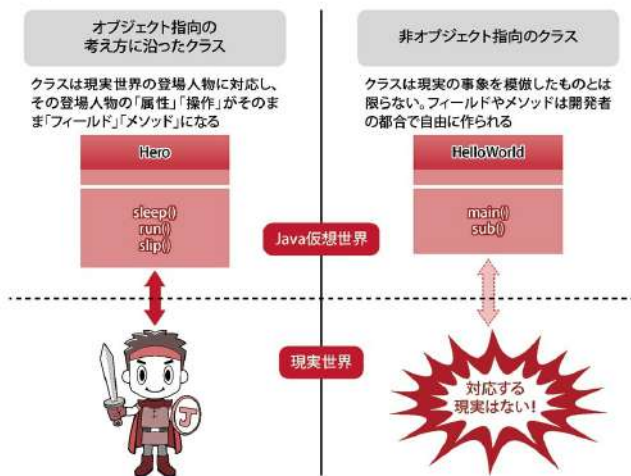



図 8-12　オブジェクト指向の考え方に沿ったクラスは現実の事柄をコードに置き換えたものである

8.5 第8章のまとめ

この章では、次のようなことを学びました。

インスタンスとクラス

- インスタンスとクラスはまったく別のものであり、混同してはならない。
- 仮想世界で活動するのは「インスタンス」（オブジェクトとも言う）。
- インスタンスを生み出すための金型が「クラス」。

フィールドとメソッド

- クラスには、属性としてフィールドを、操作としてメソッドを宣言する。
- final が付いたフィールドは、定数フィールドであり値が不変になる。
- this は「自分のインスタンス」を表すキーワードである。

クラス型

- クラスを定義することにより、そのクラス型の変数を宣言できるようになる。
- あるクラス型の変数には、そのクラスのインスタンスを格納できる。

インスタンス化

- new 演算子を用いることで、クラスからインスタンスを生み出せる。
- あるクラス型変数にインスタンスが格納されているとき、「変数名 . フィールド名」や「変数名 . メソッド名 ()」で、そのインスタンスのフィールドやメソッドを利用することができる。

8.6 練習問題

練習 8-1

現実世界の聖職者「クレリック」を表現するクラス Cleric を public で宣言してください。その際、属性や操作は宣言する必要はありません（中身が何もないクラスで構いません）。

練習 8-2

聖職者は勇者のように名前や HP を持っており、さらに魔法を使うための MP も持っています。そこで、練習 8-1 で宣言した中身がない Cleric クラスに「名前」「HP」「最大 HP」「MP」「最大 MP」を属性として追加してください。なお、HP と最大 HP は整数で初期値 50、MP と最大 MP は整数で初期値 10 であり、最大 HP と最大 MP は定数フィールドとして宣言してください。そして作成したクラスに適切なファイル名を付けてください。

練習 8-3

聖職者は「セルフエイド」という魔法を使うことができ、MP を 5 消費することで自分自身の HP を最大 HP まで回復することができます。そこで、練習 8-2 で宣言した Cleric クラスに、「selfAid()」というメソッドを追加してください。なお、このメソッドは引数なしで、戻り値もありません。

練習 8-4

聖職者は「祈る」（pray）という行動を取ることができ、自分の MP を回復できます。回復量は祈った秒数にランダムで 0 ～ 2 ポイントの補正を加えた量です（3 秒祈ったら回復量は 3 ～ 5 ポイントのいずれか）。ただし最大 MP よりも回復することはありません。

そこで、練習 8-3 で宣言した Cleric クラスに「pray()」というメソッドを追加してください。このメソッドは引数に「祈る秒数」を指定することで、戻り値として「実際に回復した MP の量」を返します。

8.7 練習問題の解答

練習 8-1 の解答

クラスの宣言に関する問題です。正解のコードは以下のとおりです。

Cleric.java
```
public class Cleric {
}
```

練習 8-2 の解答

フィールドの宣言に関する問題です。以下は解答例のコードです（おおむね合っていれば正解とします）。

Cleric.java
```
public class Cleric {
  String name;
  int hp = 50;
  final int MAX_HP = 50;
  int mp = 10;
  final int MAX_MP = 10;
}
```

このコードのファイル名は「Cleric.java」になります。

練習 8-3 の解答

メソッドの宣言に関する問題です。以下は解答例のコードです（おおむね合っていれば正解とします）。

Cleric.java
```
public class Cleric {
  String name;
```

```
3   int hp = 50;
4   final int MAX_HP = 50;
5   int mp = 10;
6   final int MAX_MP = 10;
7
8   public void selfAid() {
9     System.out.println(this.name + "はセルフエイドを唱えた！");
10    this.hp = this.MAX_HP;
11    this.mp -= 5;
12    System.out.println("HPが最大まで回復した。");
13  }
14 }
```


練習 8-4 の解答

引数と戻り値があるメソッドの宣言に関する問題です。以下は解答例のコードです（おおむね合っていれば正解とします）。

Cleric.java

```
1  import java.util.*;
2  public class Cleric {
3    String name;
4    int hp = 50;
5    final int MAX_HP = 50;
6    int mp = 10;
7    final int MAX_MP = 10;
8
9    public void selfAid() {
10     System.out.println(this.name + "はセルフエイドを唱えた！");
11     this.hp = this.MAX_HP;
12     this.mp -= 5;
13     System.out.println("HPが最大まで回復した。");
14   }
```

```
15
16    public int pray(int sec) {
17      System.out.println
            (this.name + "は" + sec + "秒間天に祈った！");
18
19      // 論理上の回復量を乱数を用いて決定する
20      int recover = new Random().nextInt(3) + sec;
21
22      // 実際の回復量を計算する
23      int recoverActual = Math.min(this.MAX_MP - this.mp, recover);
24
25      this.mp += recoverActual;
26      System.out.println("MPが" + recoverActual + "回復した。");
27      return recoverActual;
28    }
29  }
```

第9章

さまざまなクラス機構

第 8 章ではオブジェクト指向に必要な知識として、
インスタンスとクラスの基本的な使い方について学びました。
この章ではインスタンスとクラスに関する理解をさらに深めた上で、
プログラミングをより便利にしてくれる
「コンストラクタ」と「静的メンバ」について学びましょう。

CONTENTS

クラス型と参照

9.1.1 仮想世界の真の姿

先輩、「仮想世界」とか「生み出される」とかイメージはわかりましたが、コンピュータの中に「本当に仮想世界の住人がいる」わけではないと思うんです。実際はどうなっているんですか？

もっともだ。クラスに関する数々の便利な機能を学ぶ前に、ここまで学んだ内容が実行されるとき、JVMの中で何が起こっているかタネあかしをしよう。今のうちにこれを理解しておくと、後々の章での学習がグッとラクになるんだ。

ここまでの解説では、インスタンスのことを「操作と属性を持ち、コンピュータ内の仮想世界に生み出される登場人物」という概念的な表現で紹介してきました。しかし、本当に「勇者」や「お化けキノコ」のような存在がコンピュータの中に住んでいて、あれこれ活躍するわけではありません。

本書がこれまで「Java仮想世界」と表現してきたものは、**実際には「コンピュータのメモリ領域」**です。この領域は、Javaのプログラムを実行する際に、JVMが大量にメモリ領域（通常は数百MB～数GB）を使って準備するもので、**ヒープ**（heap）といいます。

そして、私たちがnewを用いてインスタンスを生み出すたびにヒープの一部の領域（通常は数十～数百バイト）が確保され、インスタンスの情報を格納するために利用されます。そのため、多くの属性を持った大きなクラスをインスタンス化すると、消費されるヒープ領域は必要とする容量に従って大きくなります。つまり次の図9-1に示すように、**インスタンスとは「ヒープの中に確保されたメモリ領域」**にすぎないのです。

図 9-1　インスタンスとは、ただのメモリ領域

せっかくコンピュータの中の楽しい世界を想像していたのに、ただのメモリ領域だったなんて…。

せっかくの夢を壊しちゃったかな？

9.1.2 クラス型変数とその内容

インスタンスの正体は「ヒープの一部に確保された単なるメモリ領域」ということがわかりました。ではそのインスタンスが生まれる際に、コンピュータの中で何が起こっているのか、次のコードを例に探っていきましょう。

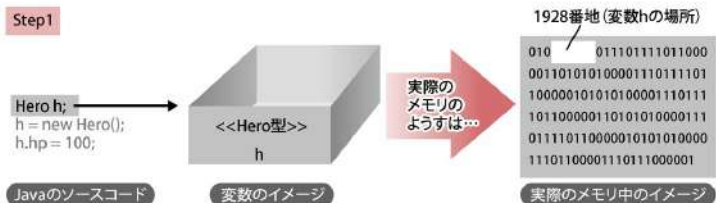
リスト 9-1　Hero クラスをインスタンス化し利用するコード

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h;
4     h = new Hero();
5     h.hp = 100;
6   }
7 }
```

9.1.3 Step1：Hero 型変数の確保

最初に動くのは、リスト 9-1 の 3 行目「Hero h;」です。この行を実行すると、JVM は「Hero 型の変数 h」をメモリ内に準備します。JVM は広いヒープ領域の中から現在利用していないメモリ領域を探し出して、自動的にそこを確保してくれます。仮に、1928 番地が空いていたので、ここが変数 h 用に確保されたとしましょう。これは次の図 9-2 のようなイメージです。

図 9-2　Hero 型の変数 h が確保された

この段階ではまだ勇者自体は生まれていません。「Hero 型のインスタンスだけを中に入れることができる」Hero 型の箱が準備されるだけです。この箱には数字や文字列を入れることはできませんし、Hero 型でない「お化けキノコ」インスタンスを入れることもできません。

菅原さん、ボクどうしても「Hero 型の箱」っていうのがしっくりこないんです。int 型や String 型だったら、数字や文字列が入るってわかりますが…。何というか、箱に「インスタンス」みたいな複雑なものが入るっていうのが考えにくいんです。

まぁそのうち慣れるよ。それに、この後の Step2 と Step3 の流れを知れば、その悩みもスッキリするはずだよ。

9.1.4 Step2：Hero インスタンスの生成

リスト 9-1 の 4 行目「h = new Hero();」は代入文です。**代入の場合は左辺より先に右辺が評価される**のでしたね。よって、Step2 では、まず「new Hero()」の部分だけを考えます。「new Hero()」が実行されると、JVM は次の図 9-3 のように

ヒープ領域から必要な量のメモリを確保します。今回は仮に「3922番地から24バイト分(3922〜3945番地)」が確保できたとします。なお、この3922番地はStep3の図9-4で出てきますので覚えておいてください。

図9-3　JVMがヒープ領域からメモリを確保し、Heroインスタンスが生成された

これで仮想世界に勇者という存在が生まれました。しかし、生まれたてのHeroインスタンスは属性が設定されていないため、まだ「名前」は空っぽ、「HP」は0です。

9.1.5 Step3：参照の代入

Step2では、リスト9-1における4行目の「h = new Hero();」の右辺について考えました。次は右辺が実行された後のことについて考えましょう。

右辺の実行が終了した後、4行目は「h = **右辺の実行結果**」という状態になります。すなわち、右辺の実行結果を変数hに代入するということが行われます。

このとき、変数hに代入される右辺の実行結果とは何でしょうか。ここまでの解説では、「Hero型の箱に、勇者インスタンスが入る」という説明をしてきました。つまり、右辺の実行結果とは勇者インスタンスなのでしょうか。

いいえ、実はそうではありません。右辺の実行結果とは、**newを実行することによって生成されたインスタンスのために確保されたメモリの先頭番地**です。今回の場合、「new Hero()」により、勇者インスタンスが3922〜3945番地に生成されているので、**変数hには3922という数字が代入**されます(次ページ図9-4)。

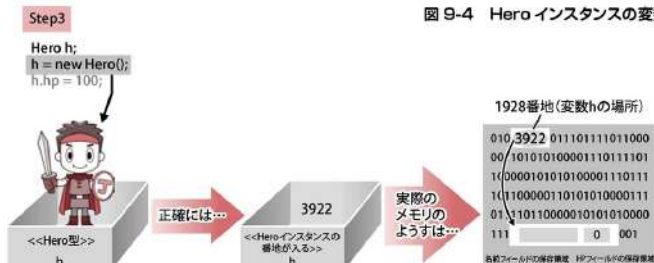



図 9-4　Hero インスタンスの変数代入

変数 h に入っている 3922 は、ただの数字にすぎません。勇者インスタンスに関する HP や名前などのさまざまな情報は変数 h の中ではなく、別のところにあります。見方を変えれば、「この変数 h には勇者インスタンスの情報の全部は入りきらないから、詳しくは 3922 番地を参照してね」とも捉えることができます。このようなことから、変数 h に入っているアドレス情報を**参照**といいます。

これって…どこかで似たようなものを学習したような気がしますけど？

そうだね、第 4 章の「配列型」と同じしくみだよ。

int[] 型や String[] 型といった「配列型」も、変数に入っているのは「実際の配列内の各データが保存されているメモリ領域の先頭番地」でしたね。Hero 型のような「クラス型」も原理は同じです。このことからクラス型と配列型は総称して「参照型」と呼ばれます。

9.1.6　Step4：フィールドへの値の代入

5 行目の「h.hp = 100;」では、変数 h に格納されている勇者の HP を 100 に設定します。この行を JVM は以下のように解釈して実行します。

①変数 h の内容を調べると、「3922 番地を参照せよ」と書かれている
②メモリ内の 3922 番地にあるインスタンスのメモリ領域にアクセスし、その中の hp フィールド部分を 100 に書き換える

図 9-5　Hero インスタンス hp フィールドへの代入

このように、①まず変数から番地情報を取り出し、②次に実際にその番地にアクセスする、という JVM の動作を**参照の解決**や**アドレス解決**といいます(図 9-5)。

インスタンスを生み出したり、フィールドにアクセスしたりするために、JVM はとても複雑なことをしているんですね。

9.1.7 同一インスタンスを指す変数

もし仮想世界に勇者が 2 人生成され、それぞれh1、h2という変数に格納されていたとしましょう。当然、勇者 h1 の hp フィールドを 10 減らしても、勇者 h2 の hp フィールドの値は減りません。同じクラスから生まれても、異なるインスタンスであれば互いに影響を受けないことを**インスタンスの独立性**といいます。

さて、以上を踏まえた上で、次のプログラムの実行結果を想像してください。

リスト 9-2　2 つの Hero 型変数

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h1;
4     h1 = new Hero();
5     h1.hp = 100;
6     Hero h2;
7     h2 = h1;
8     h2.hp = 200;
9     System.out.println(h1.hp);
```

```
10      }
11  }
```

えーっと、h1とh2の2つがあって…「インスタンスの独立性」があるから、h1のhpフィールドには100、h2のhpフィールドには200が入って…。画面に表示されるのは「100」かな？

いいや違うよ。図に書いて、よく考えてごらん。

このプログラムを正しく理解するためのポイントは7行目の「h2 = h1;」です。これは変数h1の内容をh2にコピーする式ですが、ここで9.1.5節で説明した**「勇者インスタンスhのために確保してあるメモリの先頭番地」**を思い出してください。ここでコピーされているのは「勇者インスタンスそのもの」ではありません。**「3922」などの番地情報**です（図9-6）。

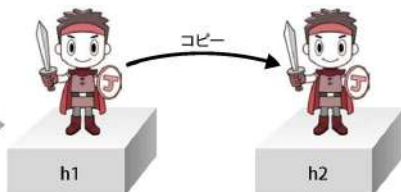
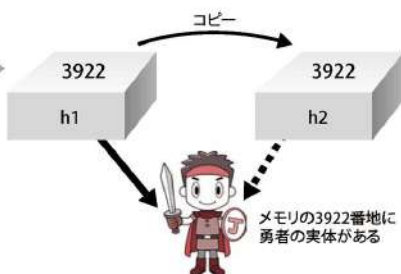



図9-6　「h2 = h1;」の動作

代入の結果、h1 と h2 の両方に番地情報の 3922 が入ります。ということは、**h1 と h2 はどちらも「まったく同じ 1 人の勇者インスタンス」を指している**のです。そのため h1 の hp フィールドへ代入しても、h2 の hp フィールドへ代入しても、**結局は同じ勇者インスタンスの HP に代入することになる**のです。

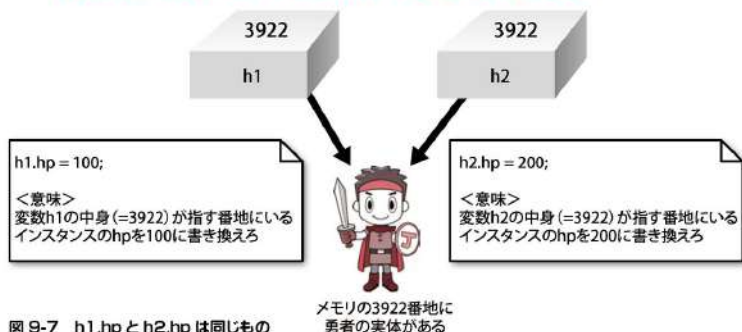



図 9-7　h1.hp と h2.hp は同じもの

リスト 9-2 に示したプログラムの場合、勇者インスタンスの hp フィールドへは h1 経由で 100 が代入されます (5 行目「h1.hp = 100;」) が、その後に同じ hp フィールドには h2 経由で 200 が上書き (8 行目「h2.hp = 200;」) されます。よって 9 行目で表示されるのは「200」なのです (図 9-7)。

そうか、「インスタンスの独立性」というのは勇者インスタンスが 2 人いた場合の話であって、今回のように「勇者が 1 人しかいない」場合には関係ないですね。h1 と h2 があったから、つい「勇者が 2 人いる」と勘違いしました。

わかった！ そもそも、リスト 9-2 の中には、**new が 1 つしかない**のよ。ということは仮に変数がいくつあっても、どんな複雑なプログラムであっても、**「仮想世界には 1 人の勇者しか生まれてない」はず**よね。

すばらしいね、そのとおりだ。いくつかの特殊な例を除いて、基本的にインスタンスを生み出す方法は new しかない。「new Hero()」した回数が勇者の人数だ。

9.1.8 クラス型をフィールドに用いる

先輩、私もう１つ違うことに気づきました。Hero クラスを定義すると Hero 型の変数を使えるようになるんですよね。ということは、もしフィールドに…。

朝香さんは本当にせっかちだね。うん、想像のとおりだよ。

朝香さんが気づいたのは、たとえば次のようなコードが書けるのではないかということです。

リスト 9-3　Sword 型フィールドを持つクラス

Sword.java

```java
// まず、Swordクラスを定義しておく
public class Sword {
  String name;     // 剣の名前
  int damage;      // 剣の攻撃力
}
```

Hero.java

```java
// 次にHeroクラスを定義する
public class Hero {
  String name;
  int hp;
  Sword sword;     // 勇者が装備している剣の情報
  void attack() {
    System.out.println(this.name + "は攻撃した！");
    System.out.println("敵に5ポイントのダメージをあたえた！");
  }
}
```

Hero クラスに新しく追加されたフィールド「sword」は、int 型や String 型ではなく Sword 型です。このように、フィールドにクラス型の変数を宣言すること

もできます。なお、今回の例のように「あるクラスが別のクラスをフィールドとして利用している関係」を **has-a の関係**と言い、次のような図で表すことがあります。

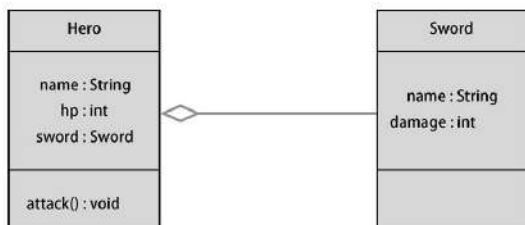



図 9-8　クラス図における has-a の関係（集約関係）

「has-a」と呼ぶ理由は、次のような英文が自然に成立するからです。

Hero has-a Sword（勇者は剣を持っている）

さて、実際にリスト 9-3 を利用する Main クラスは、次のようなものになるでしょう。

リスト 9-4

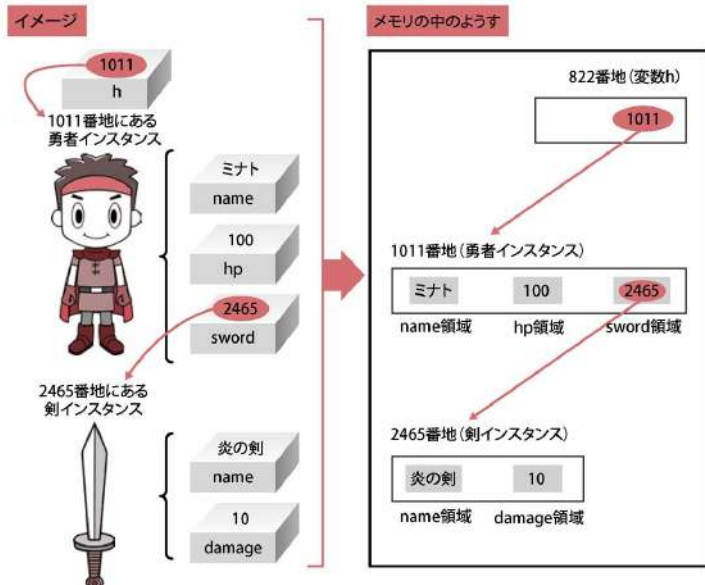
Main.java

```
public class Main {
  public static void main(String[] args) {
    Sword s = new Sword();
    s.name = "炎の剣";
    s.damage = 10;
    Hero h = new Hero();
    h.name = "ミナト";
    h.hp = 100;
    h.sword = s;
    System.out.println("現在の武器は" + h.sword.name);
  }
}
```

sword フィールドに生成済みの剣インスタンス（の番地）を代入

勇者「の」剣「の」名前

10行目は少し複雑ですが、このときのメモリのようすをしっかりイメージしてみましょう。次の図9-9を見てください。

図9-9　has-a関係にあるインスタンスのイメージ

822番地にある変数hには、勇者インスタンスのアドレス情報（1011番地）が入っています。そして勇者インスタンスに含まれるsword領域には、剣インスタンスのアドレス情報（2465番地）が格納されています。すなわち先ほどの「Hero has-a Sword」の関係が成立しており、HeroクラスがSwordクラスをフィールドとして利用していることがわかります。

9.1.9 クラス型をメソッド引数や戻り値に用いる

クラス型はフィールドの型として用いることができるだけではなく、メソッドの引数や戻り値の型として利用することもできます。例として、すでにある勇者クラスに加え、魔法使い(Wizard)のクラスを作ってみましょう。魔法使いは、勇者のHPを回復させる魔法(heal)を使うことができます。

リスト 9-5

Wizard.java

```
1 public class Wizard {
2   String name;
3   int hp;
4   void heal(Hero h) {
5     h.hp += 10;
6     System.out.println(h.name + "のHPを10回復した!");
7   }
8 }
```

引数は Hero 型

勇者の HP に 10 を加える

heal() メソッドが呼び出されると、魔法使いインスタンスは勇者のHPを10回復させます。ただし、仮想世界には勇者が2人以上生み出されている(=2回以上newされている)可能性もありますから、呼び出されるときに**「どの勇者を回復させるのか」を引数hとして受け取る**必要があります(リスト9-5の4行目)。実際に、このWizardクラスを利用したプログラムは以下のようになります。

リスト 9-6

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h1 = new Hero();
4     h1.name = "ミナト";
5     h1.hp = 100;
6     Hero h2 = new Hero();
```

```
 7        h2.name = "アサカ";
 8        h2.hp = 100;
 9        Wizard w = new Wizard();
10        w.name = "スガワラ";
11        w.hp = 50;
12        w.heal(h1);    // ミナトを回復させる（HP: 100 → 110）
13        w.heal(h2);    // アサカを回復させる（HP: 100 → 110）
14        w.heal(h2);    // アサカを回復させる（HP: 110 → 120）
15    }
16 }
```

9.1.10 String 型の真実

ではこの節の最後に、文字列を利用しているとき JVM の中で何が起こっているか、タネあかしをしよう。

第 1 章から登場していた String 型ですが、実は **int 型や double 型の仲間ではなく、Hero 型と同じ「クラス型」**です。これまで、「String 型変数の中には文字列がそのまま入っている」と解釈していたかもしれませんが、本当の姿は次の図 9-10 ようなものです。

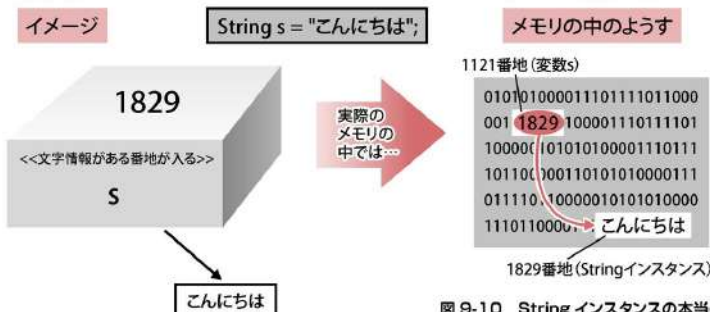



図 9-10　String インスタンスの本当の姿

しかし、疑問も残ります。Hero 型や Sword 型は私たち自身が Hero クラスや Sword クラスを定義したので利用可能になりました。しかし、String クラスを定

義した覚えはないにも関わらず私たちはString型を利用できているのはなぜでしょうか？　この答えはAPIリファレンスが明らかにしてくれます。Mathクラスや Systemクラスのように APIとして標準添付されている膨大な数のクラスの中にStringクラス（正式名称はjava.lang.Stringクラス）が含まれています。

図9-11　StringクラスのAPIリファレンス

java.lang

クラス String

java.lang.Object
　└ java.lang.String

すべての実装されたインタフェース:
　Serializable, CharSequence, Comparable<String>

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence

Stringクラスは文字列を表します。Javaプログラム内の「abc」などのリテラル文字列はすべて、このクラスのインスタンスとして実行されます。

私たちがこれまでString型を「int型と似たようなもの」として扱い続け、本章に至るまで「実はJavaが準備してくれていたクラスを利用していた」と気づかなかった（気づかずに済んだ）理由は、Javaという言語が作られるときに、次のような特別の配慮がなされたためです。

java.langパッケージに宣言されている：

第6章で紹介したとおり、java.langパッケージに所属するクラスを利用する際は、特例としてimport文を記述する必要がありません。この特例のおかげで、本来「java.lang.String s;」と宣言する必要があるところを、単に「String s;」と書けば利用できるようになっています。

二重引用符で文字列を囲めばインスタンスを生成・利用できる：

通常、インスタンスを生成するにはnew演算子を利用する必要があります。しかし文字列はプログラムの中で多用されるため、その都度newを書いていてはソースコードが「newだらけ」になってしまいます。そこで、「二重引用符で文

字列を囲めば、その文字列情報を持ったStringインスタンスを利用できる」という特例が設けられました。この特例のおかげで、new演算子を使うことなく「String s = "こんにちは";」というシンプルな記述が可能になっています。

実は、Stringもクラスには違いないのですから、HeroやSwordと同じように「newでインスタンス生成」することもできます（リスト9-7）。ただし、この方法は効率が悪いので、通常は利用しないでください。

リスト9-7

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String s = new String("こんにちは");
4      System.out.println(s);
5    }
6  }
```

画面に「こんにちは」と表示される

ん？　このコードのnewって少し文法がおかしくありませんか？　今までの書き方だと「new String();」と書くのでは？

そうだね。実はStringクラスは**「newするとき、ついでに追加情報を指定できる特別なしくみ」**に対応しているんだ。Stringはインスタンスが生成された直後に、この追加情報（＝「こんにちは」）を自分自身の中に書き込むことができるんだよ。

ひょっとして、その「追加情報付きでnewできるしくみ」はボクたちのHeroクラスやSwordクラスでも使えますか？

うん。とても便利な機能だから、次の節でしっかり学習しよう。

9.2 コンストラクタ

9.2.1 生まれたてのインスタンスの状態

前節をマスターした私たちは、Javaの仮想世界にインスタンスを自由に生み出して利用することができるようになりました。しかし、実際にクラスを用いたプログラミングをし始めると、ある「めんどうさ」を感じるようになるはずです。その例として、9.1.9項に登場したリスト9-6を再び次に示します。

リスト9-6 （抜粋）

```
1   ⋮
2   Hero h1 = new Hero();
3   h1.name = "ミナト";
4   h1.hp = 100;
5   Hero h2 = new Hero();
6   h2.name = "アサカ";
7   h2.hp = 100;
8   Wizard w = new Wizard();
9   w.name = "スガワラ";
10  w.hp = 50;
11  w.heal(h1);
12  w.heal(h2);
13  w.heal(h2);
14  ⋮
```

コメントで示したように、newでインスタンスを生成した直後、必ずフィールドの初期値を代入しています。なぜなら**newで生み出されたばかりのインスタンスのフィールド（nameやhp）には、まだ何も入っていないから**です。厳密にいえば、各フィールドには値が「入っていない」わけでなく、次のような初期値が設定されています。

表9-1 フィールドの初期値

int型, short型, long型 等の数値の型	0
char型（文字）	¥u0000
boolean型	false
int[] 型などの配列型	null
String型などのクラス型	null

生まれたばかりの勇者のHPって0なんですね。こんな状態で、そのまま冒険に出たら大変だ…。

すぐに死んじゃうね（笑）。

9.2.2 フィールド初期値を自動設定する

こらぁミナトーっ！ やっと見つけた（怒）！

…ど、どうしたの朝香さん。そんなに息切らして。

あなたの作ったHeroクラスを使ってプログラムを組んだら、ゲーム開始直後にHPが0になってるの！ いきなり死んでるじゃない。こんなバグだらけのHeroクラス、使わせるんじゃないわよ！

勇者の初期 HP は 100 って決まってるんだ。だから朝香さんのほうで new した後に 100 を代入してよ。

そんなの今、初めて聞いたわよ！ っていうか**「勇者として初期 HP=100 と決まってる」**なら自動的にそうなるように **Hero クラス側で責任を持ちなさい**よ！

まぁまぁ落ち着いて。そんなときのために最適な Java のしくみがあるよ。

湊くんが制作している RPG では、どうやら生まれたばかりの勇者の HP は常に 100 とする決まりのようです。実際に前節のリスト 9-6 では、「ミナト」「アサカ」の 2 人の勇者は共に、new の直後に HP の初期値として 100 を代入していました。

しかし、実際の開発現場において、特にゲームなど大規模な開発を行う場合、1 人ですべてを開発することは、まずありえません。そのためクラスを作るにあたっては、自分以外の開発者が Hero クラスを利用することも考えておかなければなりません。そして、その**開発者が正しく HP に 100 を代入してくれるとは限らないのです**。朝香さんのように、うっかり初期化し忘れるかもしれませんし、Hero クラスを作った人が想定しないような数、たとえば負の数や非常に大きな数で初期化してしまうかもしれません。これではゲームは正しく動作しないでしょう。

「勇者の初期 HP が 100 であること」は、仮想世界における勇者自身に関することですし、Hero クラスの開発者が一番よく知っていることです。逆に、Hero クラスを使う側にとっては、「初期値が何であるか」は知らないのが当然のこととも言えます。ですから「Hero クラスを作る側で責任を持つべきこと」という朝香さんの主張はもっともな話です。

このような場合に備え、Java では**「インスタンスが生まれた直後に自動実行される処理」をプログラミングできる**ようになっています。次ページのリスト 9-8 を見てください。

リスト 9-8

Hero.java

```
public class Hero {
  int hp;
  String name;
    ：
  void attack() {
    ：
  }
  Hero() {
    this.hp = 100;    // hpフィールドを100で初期化
  }
}
```

「new された直後に自動的に実行される処理」を書いたメソッド

このクラスには 8 行目から Hero() というメソッドが追加されています。attack() などの通常のメソッドは「誰かから呼ばれないと動かない」ものですが、この Hero() メソッドだけは、「このクラスが **new された直後に自動的に実行される**」という特別な性質を持っています。このようなメソッドを**コンストラクタ**(constructor) と呼びます。上記の Hero() メソッドはコンストラクタとして定義されており、new されると自動的に実行されて HP に 100 が代入されます。そのため main メソッド側で HP に初期値を代入する必要はありません。

リスト 9-9

Main.java

```
public class Main {
  public static void main(String[] args) {
    Hero h = new Hero();


    System.out.println(h.hp);
  }
```

この指示によりインスタンスが生まれ、さらにコンストラクタの働きで HP に 100 が代入される

すでに hp には 100 が代入されているので、画面に「100」と表示される

```
8 }
```

よかった！　これで誰が勇者インスタンスを生成しようと、生まれたばかりの勇者は、必ず HP が 100 になりますね！

図 9-12　new をするだけで自動実行される処理

ここで意識しておいてほしいのは、**「コンストラクタは、私たちプログラマが直接呼び出すものではない」**という点です。私たちが直接行うことはあくまでも「Hero h = new Hero();」でインスタンスを生成することであって、それによって**間接的に Hero() が実行**されるのです。私たちが main メソッドなどの中から「h.Hero();」のようにコンストラクタを直接呼び出すことはできません。

そういえば現実世界における「赤ちゃん」も、生まれた直後に自分から泣き出しますよね。

そうだね。もし人間にコンストラクタが定義されているとしたら、そこには「泣く ();」って書いてあるだろうね。

9.2.3 コンストラクタの定義方法

先輩、このクラスには attack() など、ほかにもたくさんメソッドがあるのに、なぜ Hero() メソッド**だけ**が自動実行されるんですか？

それは Hero() メソッドだけが「**自動実行されるメソッドの条件**」を満たしているからだよ。

一見すると、Hero() メソッド（＝コンストラクタ）も、ほかのメソッドと違いがないように見えます。しかし、new でインスタンスを生成したときに自動実行されるのは Hero() だけです。実は Java では、クラスに記述されているメソッドのうち、以下の条件をすべて満たすメソッドだけがコンストラクタと見なされ、自動実行される決まりになっています。

コンストラクタと見なされる条件

①メソッド名がクラス名と完全に等しい
②メソッド宣言に戻り値が記述されていない（void もダメ）

Hero() がコンストラクタとして実行されたのは、「Hero クラス」の中に「Hero()」という完全に同名で定義されており、その戻り値が記述されていないからです。

コンストラクタの基本書式

```
1 public class クラス名 {
2   クラス名() {
3     ：
4   }
5 }
```

ここに自動実行処理を記述する

フィールド宣言による初期値の定義

フィールドの値を固定の値に初期化するだけでよければ、コンストラクタを用いなくても、フィールド宣言時に代入式を書くことでも実現可能です。しかし、次項で説明するような複雑な初期化を行いたい場合は、コンストラクタを使わなければ実現できません。

9.2.4 コンストラクタに情報を渡す

自動的に勇者のHPが100になったのは嬉しいんですが、「名前」は自動的に代入できないんですかねぇ？

new で生み出す勇者全員に「ミナト」って同じ名前が入っちゃうじゃない？ でも勇者の名前って、それぞれ違うわよね？

HPフィールドは「100」という固定の値で初期化すればよいため、単純なコンストラクタで済みました。しかし、勇者の名前は生み出すインスタンスによって異なるはずです。このような場合は次のリスト9-10のように、コンストラクタが「毎回異なる追加情報」を引数で受け取れるように宣言することができます。

リスト 9-10　コンストラクタで引数を追加情報として受け取る

Hero.java

```
1  public class Hero {
2    ⋮
3    Hero(String name) {        ← 引数として文字列を1つ受け取る
4      this.hp = 100;
5      this.name = name;      // 引数の値でnameフィールドを初期化
6    }
7  }
```

9章

これで、Hero クラスは **new するときに名前の初期値も指定できる**ようになりました。

でも菅原さん、私たちはコンストラクタを「直接呼び出せない」はずですよね？（9.2.2 項参照）。メソッドのように呼び出すときに引数を渡せないとしたら、どうやって引数を渡せばいいんですか？

いい質問だね。代わりに new するときに渡しておくんだよ。

このような Hero クラスを利用する場合は、**コンストラクタに渡すべき引数を new する際に指定**します。次のリスト 9-11 では、new をした時点で与えられた「ミナト」という引数が、コンストラクタ Hero() が自動実行される際にパラメータとして渡されます。

リスト 9-11　new する際に引数を渡す

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h = new Hero("ミナト");
4
5     System.out.println(h.hp);
6     System.out.println(h.name);
7   }
8 }
```

- 3行目：こう書いておけばコンストラクタには「ミナト」が渡される
- 5行目：100 と表示される
- 6行目：ミナトと表示される

この処理のようすを図で表すと、次の図 9-13 のようになります。

図 9-13　new 時に指定した引数が、コンストラクタ実行時に利用される

9.2.5　2つ以上の同名コンストラクタを定義する

Hero にコンストラクタができて「new Hero(" ミナト ");」のようにできるようになったのは確かに便利です。ただ、簡単な動作テストなど「別に名前はどうでもいい」ときもあって、単に「new Hero();」としたい場合にはどうすればいいんでしょうか？

なるほど。そういうときに、めんどうだから「new Hero();」としたいのに、引数がないためにエラーが出てしまうんだね。

現在の Hero クラスには、「文字列引数を 1 つ受け取るコンストラクタ」が定義されています。そしてコンストラクタが「new された際に必ず自動的に実行されるもの」である以上、「new をする側としては、必ず引数となる文字列を 1 つ与える」必要があります。

つまり、このコンストラクタを作ったことによって、**インスタンスを生成するときには、必ず名前を指定する必要が生じた**わけです。試しに、朝香さんの言うように、引数なしで「new Hero();」を実行するとエラーになります。

この問題は**「引数を受け取らないコンストラクタ」も同時に定義することで解決**できます。次ページのリスト 9-12 を見てください。

リスト 9-12　コンストラクタのオーバーロード

Hero.java

```
public class Hero {
  ⋮
  Hero(String name) {        // 以前からあったコンストラクタ①
    this.hp = 100;
    this.name = name;
  }
  Hero() {                   // 新しく作ったコンストラクタ②
    this.hp = 100;
    this.name = "ダミー";     // 新ダミーの名前を設定する
  }
}
```

先輩。これって、第 5 章で習ったオーバーロード（5.4 節）ですよね？

そのとおりだよ。「同じ名前だが引数が異なるメソッドを複数定義」するオーバーロードは、コンストラクタでも可能なんだ。

ということは、実行時に「どちらが動くか」は Java が空気を読んで判断してくれるんですね。

複数のコンストラクタが定義されていた場合

new するときに渡した引数の型・数・順番に対応するコンストラクタが動作する（複数のコンストラクタが定義されていても、1 つだけしか動作しない）。

コンストラクタのオーバーロードを実際に活用する例が、次のリスト 9-13 です。

リスト 9-13　コンストラクタをオーバーロードしたクラスの利用

Main.java

```
public class Main {
  public static void main(String[] args) {
    Hero h1 = new Hero("ミナト");

    System.out.println(h1.name);
    Hero h2 = new Hero();

    System.out.println(h2.name);
  }
}
```

文字列引数があるのでコンストラクタ①が呼び出される

画面に「ミナト」と表示される

引数がないのでコンストラクタ②が呼び出される

画面に「ダミー」と表示される

9.2.6 暗黙のコンストラクタ

ところで、コンストラクタが定義されていない Hero クラスは「new Hero();」で生成できたのに、引数ありコンストラクタを定義しただけで同様のことが不可能になったのはなぜでしょうか。

実は Java では、すべてのクラスはインスタンス化に際して**必ず何らかのコンストラクタを実行すること**になっています。ですから、**本来すべてのクラスは、**「引数のない、何も処理をしないコンストラクタ」でよいので、**最低でも 1 つ以上のコンストラクタ定義を持っていなければなりません。**コンストラクタが 1 つも定義されていないクラスは許されないのです。

えぇ〜！ そんなのめんどうですよ。次に開発する予定の「宝の地図」クラス（Map クラス）はインスタンス化の直後に初期化する必要はないんです。

ルールに従うためだけに中身のないコンストラクタを定義しないとならないとしたら不便ね。

リスト 9-14 すべてのクラスは必ずコンストラクタを定義しなければならない？

Map.java

```
1  public class Map {
2    ⋮
3    Map() {
4    }
5  }
```

new 時に自動実行したいことは何もないが、しかたなくダミーのコンストラクタを定義

上記のリスト 9-14 のように、わざわざダミーでコンストラクタを定義するのはめんどうですので、Java では以下のような特例を設けています。

コンストラクタの特例

クラスに 1 つもコンストラクタが定義されていない場合に限って、「引数なし、処理内容なし」のコンストラクタ（デフォルトコンストラクタ）の定義がコンパイル時に自動的に追加される。

これまでサンプルで示してきた Hero クラスは、**この特例によって引数なしコンストラクタがこっそり自動定義されたため、「new Hero();」によるインスタンス化が可能だった**のです。ですが、新たに引数を 1 つ含むコンストラクタを定義した時点でこの特例は適用除外となり、「new Hero();」という記述によるインスタンスの生成はできなくなったのです。

9.2.7 ほかのコンストラクタを呼び出す

菅原さん、コンストラクタを複数作っていて、少し「気持ち悪い」ことがあるんですけど。

「複数のコンストラクタに、同じ処理をいくつも書いてること」だね？

2つ以上のコンストラクタを定義していると、重複する処理を記述することもあるでしょう。たとえば、9.2.5項のリスト9-12をもう一度見てください。コンストラクタ①とコンストラクタ②の内容には重複があります。

2つのコンストラクタは、どちらもHPに100を代入しています。しかし、もし将来「初期HPを200に変更する」ことになったら、コンストラクタ①と②の両方を修正する必要が生じます（この例は極めてシンプルですが、本格的なプログラムにおいて、重複部分はさらに多くなることが一般的です）。

そこで思いつくのが、コンストラクタ②の中で、コンストラクタ①を呼び出す方法です。たとえば、次のリスト9-15のようなコードでHPへの代入を1か所に集中させようと考えるかもしれません。

リスト9-15　コンストラクタの中から別のコンストラクタを呼び出す（エラー）

Hero.java

```
1  public class Hero {
2    ⋮
3    Hero(String name) {
4      this.hp = 100;
5      this.name = name;
6    }
7    Hero() {     // コンストラクタ②
8      this.Hero("ダミー");
9    }
10 }
```

コンストラクタ①（3〜6行目）

8行目：コンストラクタ①を呼び出したいが、この行はエラーになるのでダメ！

しかし残念ながら、このコードはコンパイルエラーになります。なぜなら、Java ではコンストラクタを直接呼び出すことができないからです。ただし、この制限には例外があります。それは**「専用の文法を用いて、コンストラクタの先頭で別のコンストラクタを呼び出す場合に限って」特別に許される**ということです。

別コンストラクタの呼び出しに関するルール

「this. クラス名 (引数);」と記述することはできない。
その代わりに「this(引数);」と記述する。

よって、リスト 9-15 の Hero クラスを正しく記述するには、次のリスト 9-16 のようにします。

リスト 9-16　コンストラクタの中から別のコンストラクタを呼び出す（正常に動作）

```
3    Hero(String name) {        ← コンストラクタ①
4      this.hp = 100;
5      this.name = name;
6    }
7    Hero() {                   ← コンストラクタ②
8      this("ダミー");           ← コンストラクタ①を呼び出す。
9    }
```

ちなみに、今回学んだ this() は、今まで利用してきた this と見た目が似ているが、何の関係もない別物と考えたほうがいい。「this. メンバ名」の this は自分自身のインスタンスを表すもの。「this(引数)」の this() は同一クラスの別コンストラクタを呼び出すためのものだ。

9.3 静的メンバ

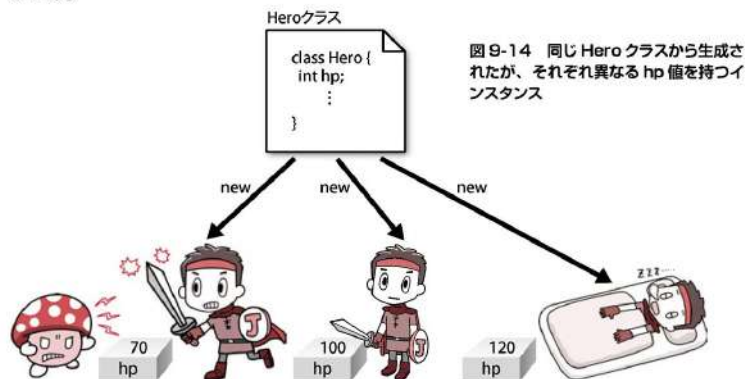
9.3.1 クラス上に準備されるフィールド

前節までで、この章で最も大切なコンストラクタの解説は終わりだよ。ところで「インスタンスの独立性」について学んだのは覚えているかな？

はい。別々の new で生まれた各インスタンスのフィールドの中身は別、ということですよね。

そうだ。第9章の締めくくりに、その独立性の例外を紹介しておこう。

new によって生成される個々のインスタンスは基本的に独立した存在です(9.1.7 項「インスタンスの独立性」参照)。よって、各勇者インスタンスが持つ同名のフィールド hp には、それぞれ別の値を格納することができます(図 9-14)。

図 9-14　同じ Hero クラスから生成されたが、それぞれ異なる hp 値を持つインスタンス

たとえば勇者3名がチーム（パーティ）を組んで冒険するRPGを作る場合を考えましょう。次のようなクラスから生成されるインスタンスでは、それぞれが名前フィールド（name）やHPフィールド（hp）、そして所持金フィールド（money）を別々に持つことになります。

リスト9-17　同じクラスから作られても、個々のインスタンスは別々のフィールドを持つ

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    int money;
5      ⋮
6  }
```

しかし、プログラムを開発していると、「各インスタンスで共有したい情報」が出てくることがあります。たとえばRPGなら、チームを組んで行動しているので「チーム全員で1つのお財布」を設定したい場合もあるでしょう。

各Heroインスタンスごとの財布は不要で、すべての勇者で1つの財布を共有すればいいってことですね。

そのとおりだよ。財布は「インスタンスに1つ」ではなく、「何に対して1つ」あればいいのかな？

「すべての勇者で1つ」つまり「Heroというクラスに対して1つ」ですね。

このように同じクラスから生成されたインスタンスでフィールドを共有したい場合には、フィールド宣言の先頭に static キーワードを追加します。

リスト 9-18　static キーワードによるフィールドの共有

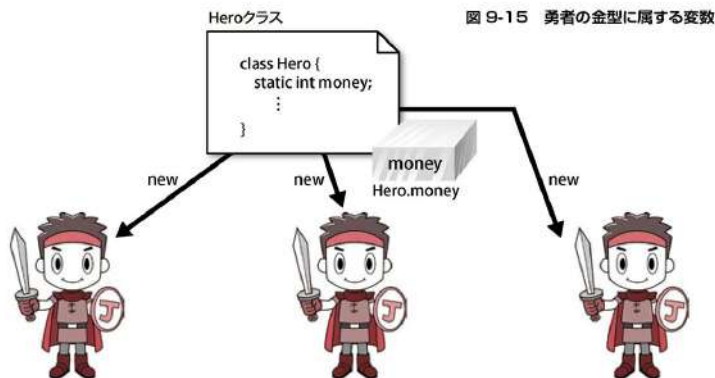
Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    static int money;   ← 静的フィールド
5      ：
6  }
```

static キーワードを指定したフィールドは特に**静的フィールド**（static field）といわれ、下記のような 3 つの特殊な効果をもたらします。

1. フィールド変数の実体がクラスに準備される

通常、フィールドが格納される箱（領域）は個々のインスタンスごとに用意されますが、静的フィールドの箱はインスタンスではなく、クラスに対して 1 つだけ用意されます。イメージで考えるならば、図 9-15 のように「勇者の金型」の上に money の箱が準備されるというイメージになります。

図 9-15　勇者の金型に属する変数

この Hero クラスに準備された箱（静的フィールド「money」）を読み書きするには、「Hero.money」という表記を使います。

静的フィールドへのアクセス方法

クラス名 . 静的フィールド名

リスト 9-19　静的フィールド money へのアクセス

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h1 = new Hero();
4      Hero h2 = new Hero();
5      System.out.println(h1.hp);
6      System.out.println(Hero.money);
7      ⋮
```

インスタンス h1 の箱 hp を表示

クラス Hero の箱 money を表示

2. 全インスタンスに、箱の分身が準備される

共通財産である金額が格納される変数（Hero.money）は、あくまでも金型に作られます。しかし同時に、h1 や h2 といった各インスタンスにも money という名前で「箱の分身」が準備され、金型の箱の別名として機能するようになります。つまり、「h1.money」や「h2.money」という分身の箱に値を代入すれば、本物の箱 Hero.money にその値が代入されるのです。

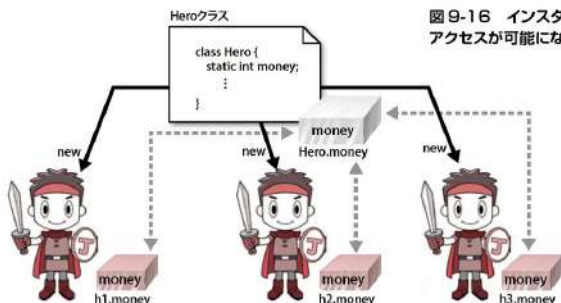



図 9-16　インスタンス経由でのアクセスが可能になる

実質的に「h1.money」「h2.money」「Hero.money」はどれも同一の箱を指すことになるんですね。

静的フィールドへの別名によるアクセス

「インスタンス変数名.静的フィールド名」と書いてもよいが、「クラス名.静的フィールド名」と同じ意味になる。

リスト 9-20　個々のインスタンスから静的フィールド money へアクセスして内容を表示

Main.java

```
public class Main {
  public static void main(String[] args) {
    Hero h1 = new Hero();
    Hero h2 = new Hero();
    Hero.money = 100;
    System.out.println(Hero.money);   // 100と表示
    System.out.println(h1.money);     // 100と表示
    h1.money = 300;                   // h1.moneyに300を代入
    System.out.println(h2.money);     // h2.moneyでも300と表示
  }
}
```

このような状態は、「h1 と h2 が money フィールドを共有している」と考えることもできるので、「静的フィールドを用いれば、インスタンス間でフィールドを共有できる」と解説されることもあります。

3. インスタンスを 1 つも生み出さなくても箱が利用可能になる

「Hero.money」は金型の上に作られる箱です。よって、まだ 1 つの実体（インスタンス）も生み出されていない状況であっても利用することができます。

リスト 9-21　インスタンスが生成されていなくても静的フィールドにアクセスできる

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     // 1人も勇者を生み出していない状況で…
4     Hero.money = 100;
5     System.out.println(Hero.money);
6   }
7 }
```

なお、静的フィールドはクラス（金型）にフィールド（箱）が所属するという特徴から、**クラス変数**と言われることもあります。

なんだか static って、ちょっと複雑な効果があるキーワードですね。

慣れれば大したことはないんだけども、もし混乱しそうなら理解は後回しでもいいよ。確かに少しややこしい割には、実際の開発では頻繁に使われるものではないんだ。それよりはコンストラクタの理解のほうが何倍も大事だよ。

public static final コンビネーション

多くの場合、static は final（第 1 章 1.3.5 項）や public（10 章で学習）と一緒に指定され、「変化しない定数を各インスタンスで共有するため」に利用されます。

9.3.2 静的メソッド

先輩、static といえば私たちが今まで使ってきた main メソッドにも付いていますよね？

そうだね。static はメソッドにも付けられるんだよ。

Hero クラスに、「勇者たちの所持金をランダムに設定する」setRandomMoney() メソッドを追加する場合、リスト 9-22 のようなコードを記述します。

リスト 9-22　静的なメソッドの例

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    static int money;
5      ⋮
6    static void setRandomMoney() {
7      Hero.money = (int) (Math.random() * 1000);
8    }
9  }
```

static を付けたメソッド

static キーワードが付いているメソッドは、**静的メソッド**（static method）または**クラスメソッド**（class method）と呼ばれ、静的フィールドとあわせて**静的メンバ**（static member）と総称されます。静的メソッドを定義すると静的フィールドと同様に以下の 3 つの効果が現れます。

1. メソッド自体がクラスに属するようになる

静的メソッドは、その実体が各インスタンスではなくクラスに属し、「クラス名 . メソッド名 ();」で呼び出せるようになります。

2. インスタンスにメソッドの分身が準備される

静的メソッドは、「インスタンス変数名 . メソッド名 ();」でも呼び出せるようになります。

3. インスタンスを 1 つも生み出すことなく呼び出せる

静的メソッドは、1 つもインスタンスを生み出していない状況であっても、呼び出すことができます。

リスト 9-23　静的メソッドの呼び出し

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero.setRandomMoney();
4     System.out.println(Hero.money);   ← ランダムな金額が表示
5     Hero h1 = new Hero();
6     System.out.println(h1.money);   ← 同じ額を表示
7   }
8 }
```

ここまで説明すれば、**main メソッドがなぜ static でなければならないのか**、想像つくんじゃないかな？ ヒントは「3 番目の効果」だ。

main メソッドが呼び出されるとき、仮想世界にはまだ 1 つもインスタンスが存在していないからですね。main メソッドが属するクラス（Main クラスなど）さえもまだインスタンス化されていない状況で、main は呼び出される必要がありますから。

9.3.3 静的メソッドの制約

実は静的メソッドの利用には重要な制約があるんだ。これを忘れて開発時に悩んでしまう人も少なくない。

静的メソッドの中に記述するコードでは、**staticが付いていないフィールドやメソッドは利用できません**。次のリスト9-24を見てください。

リスト9-24 静的なメソッド中でアクセスできるのは静的メンバだけ

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4    static int money;
5    :
6    static void setRandomMoney() {
7      Hero.money = (int) (Math.random() * 1000);
8      System.out.println(this.name + "たちの所持金を初期化しました");
9    }
10 }
```

エラー

静的メソッドsetRandomMoney()の内部である8行目で、フィールドnameへアクセスしようとしていますが、この処理はエラーとなります。

静的メソッドsetRandomMoney()は、**まだ1つも勇者インスタンスが存在しない状況でも呼び出されることがあるメソッド**です。もし仮想世界に1つも勇者インスタンスがない状況でsetRandomMoney()が動いてしまったら、「自分自身のインスタンス」のメンバである「this.name」をうまく処理できないことは明らかです。ですから静的メソッド内部では、静的メンバしか利用できないことになっているのです。

9.4 第9章のまとめ

この章では、次のようなことを学びました。

クラス型と参照

- クラス型変数の中には、「インスタンスの情報が格納されているメモリ番地」が入っている。
- あるクラス型変数を別変数に代入すると、番地情報だけがコピーされる。
- クラス型は、フィールドやメソッドの引数・戻り値の型としても利用できる。

コンストラクタ

- 「クラス名と同一名称で、戻り値の型が明記されていないメソッド」はコンストラクタとして扱われる。
- コンストラクタは、new によるインスタンス化の直後に自動的に実行される。
- 引数を持つコンストラクタを定義すると、new をする際に引数を指定してコンストラクタを実行させることができる。
- コンストラクタはオーバーロードにより複数定義できる。
- クラスにコンストラクタ定義が1つもない場合に限って、コンパイラが「引数なし・処理内容なし」のデフォルトコンストラクタを自動定義してくれる。
- this () を用いれば、同一クラスの別コンストラクタを呼び出すことができる。

静的メンバ

- static キーワードが付いている静的メンバ（フィールドおよびメソッド）は、
  ① 各インスタンスにではなく、クラスに実体が準備される。
  ②「クラス名 . メンバ名」、「インスタンス変数名 . メンバ名」のどちらでも同じ実体にアクセスすることになる。
  ③ 1つもインスタンスを生み出していなくても利用可能である。
- 静的メソッドは、その内部で静的ではないメソッドやフィールドを利用することができない。

9.5 練習問題

練習 9-1

第 8 章の練習問題で作成した Cleric クラスに関して、以下の 2 つの修正を行ってください。

① 現時点の Cleric クラスの定義では、各インスタンスごとの最大 HP と最大 MP フィールドに情報を保持します。しかし、すべての聖職者の最大 HP は 50、最大 MP は 10 と決まっており、各インスタンスがそれぞれ情報を持つのはメモリのムダです。
そこで、最大 HP・最大 MP のフィールドが各インスタンスごとに保持されないように、フィールド宣言に適切なキーワードを追加してください。

② 以下の方針に従って、コンストラクタを追加してください。

A)このクラスは、new Cleric(" アサカ ", 40, 5) のように、名前・HP・MP を指定してインスタンス化することができます。

B)このクラスは、new Cleric(" アサカ ", 35) のように、名前と HP だけを指定してインスタンス化することもできます。この場合、MP は最大 MP と等しい値で初期化されます。

C)このクラスは、new Cleric(" アサカ ") のように、名前だけを指定してインスタンス化することもできます。この場合、HP と MP は最大 HP と最大 MP で初期化されます。

D)このクラスは、new Cleric() のように、名前を指定しない場合にはインスタンス化することはできないものとします（名前がない Cleric は仮想世界に生み出せない）。

E) コンストラクタは極力重複コードをなくすように記述します。

練習問題の解答

練習 9-1 の解答

(注)静的フィールド宣言とコンストラクタ宣言部のみを掲載してあります。

```
static final int MAX_HP = 50;
static final int MAX_MP = 10;

public Cleric(String name, int hp, int mp) {
  this.name = name;
  this.hp = hp;
  this.mp = mp;
}
public Cleric(String name, int hp) {
  this(name, hp, Cleric.MAX_MP);
}
public Cleric(String name) {
  this(name, Cleric.MAX_HP);
}
```

第10章

カプセル化

第8章や第9章で学んだ文法を用いれば、
クラスやインスタンスを利用して現実世界を模倣した
オブジェクト指向のプログラムを自由に開発できます。
しかし、間違えて属性を書き換えてしまったり、
誤った操作を呼び出してしまうなどのヒューマンエラーを
完全になくすことはできません。
そのため、Javaにはミスを未然に防ぐ「カプセル化」のしくみがあります。
この章では、その便利なしくみについて学んでいきましょう。

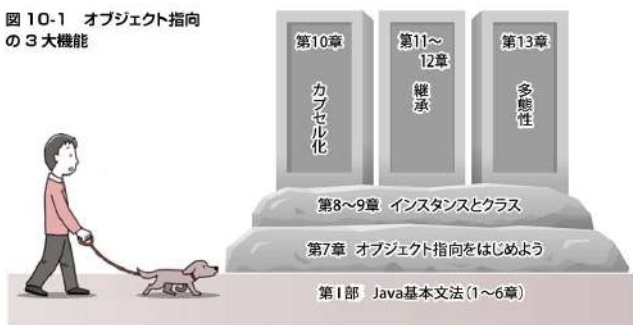
CONTENTS

10.1 カプセル化の目的とメリット

10.1.1 オブジェクト指向の３大機能

第８章と第９章で学んだ「クラスの宣言方法とインスタンスの利用法」をマスターしていれば、オブジェクト指向の考え方に沿ったプログラムを作ることができます。

ですが、もし「オブジェクト指向プログラミングを、さらにラクに、さらに楽しくするしくみ」があるとしたら、もっと魅力的だと思いませんか？ たとえば今までの半分の労力で、もっと楽しく、しかもエラーをできるだけ防げるプログラムが書けたとしたらどうでしょう。実は Java には、そのような嬉しいしくみが３つも備わっています（図 10-1）。

図 10-1　オブジェクト指向の３大機能

この図に示した「カプセル化」「継承」「多態性」は、オブジェクト指向の３大機能と呼ばれています。この章では、まず最も簡単な「カプセル化」から学びましょう。

10.1.2 カプセル化とは？

Java に備わっている「カプセル化」とは、フィールドへの読み書きやメソッド

の呼び出しを制限する機能です。たとえば、「このメソッドは、A クラスからは呼び出せるけど、B クラスからは呼び出せない」「このフィールドの内容は、誰でも読めるけど、書き換えは禁止」といったことを実現できます。

「制限して不便にしてしまう機能」なんて意味あるんですか？ 誰でも自由にメンバを利用できるほうが便利だと思うけど…。

いや、そうとも限らないんだよ。

「大切なモノに対するアクセスは不自由であるほうがよい」ことを、私たちは現実世界でもよく知っています。たとえば、あなたの大切な銀行口座に対して、誰もが出し入れ可能だとしたら、どうでしょうか。あなたが気づかないうちにあなたのお金を誰かが引き出してしまうかもしれませんね。

ほかにも「登録された人しか立ち入れないように、塀に囲まれ、門には守衛がいる軍事施設」など、私たちの周りでは「制限」が行われている例が多くあります（図 10-2）。確かに不便ではありますが、この制限があるからこそ、次のようなメリットを享受できるのです。

- **悪意のある人が軍事施設に入り、ミサイルを発射してしまうことを防げる**
- **子どもがうっかり軍事施設に入り、ミサイルを発射してしまうことを防げる**
- **万一、何者かがミサイルを発射した場合、登録された人に犯人を絞り込める**

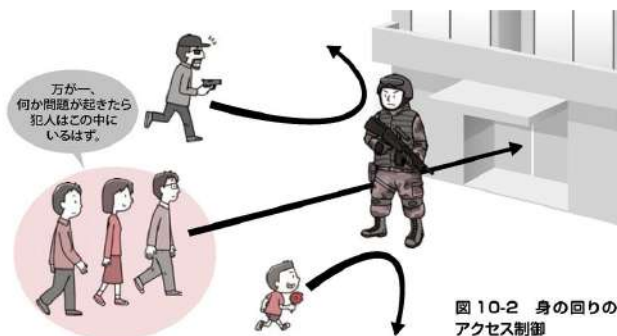



図 10-2　身の回りのアクセス制御

この例のように、情報へのアクセスや動作の実施について、「誰に何を許すか」を定めて制限することを、**アクセス制御**(access control)といいます。Javaにおけるカプセル化とは、大切な情報(フィールド)や操作(メソッド)についてアクセス制御をかけることにより、悪意や間違いによるメンバの利用を防止し、想定しない利用が発生したならば、その原因箇所を特定しやすくするためのしくみなのです。

10.1.3 アクセス制御されない怖さ

カプセル化によるアクセス制御の方法を学ぶ前に、「アクセス制御されないプログラムがいかに怖いか」を、RPGの開発プロジェクトを例に考えてみましょう。次のリスト10-1は、第9章までに作成したHeroクラスを改良したものです。

リスト10-1　アクセス制御されていないプログラムの例

Hero.java

```
1  public class Hero {
2    int hp;
3    String name;
4    Sword sword;
5    static int money;
6    void bye() {
7      System.out.println("勇者は別れを告げた");
8    }
9    void die() {
10     System.out.println(this.name + "は死んでしまった！");
11     System.out.println("GAME OVERです。");
12   }
13   void sleep() {
14     this.hp = 100;
15     System.out.println(this.name + "は眠って回復した！");
16   }
17   void attack(Matango m) {
```

```
18        System.out.println(this.name + "の攻撃！");
19        ⋮
20        System.out.println("お化けキノコ" + m.suffix
              + "から2ポイントの反撃を受けた");
21        this.hp -= 2;        反撃を受けるとHPが2減る
22        if (this.hp <= 0 ) {
23          this.die();
24        }
25      }
26      ⋮
27  }
```

この勇者はインスタンス化されるとHPが100に設定されます。そして敵との戦いでHPが減少し、0以下になったら死亡しゲームオーバーとなります。しかし、あなたはHeroクラスを使ったゲームのテスト中に、「一度もモンスターと戦っていないのに勇者のHPがマイナス100になっている」ことに気づきます（図10-3)。

図10-3　戦っていないのにHPがマイナスの勇者（死んでいる！）

あなたは数万行もあるゲームプログラムのいったいどこに不具合の原因があるのかを夜中まで調査し、原因をつきとめました。それは新入社員のAさんが開発した次のリスト10-2の「宿屋クラス」でした。

リスト 10-2　「宿屋クラス」の不具合

Inn.java

```
1  public class Inn {
2    void checkIn(Hero h) {
3      h.hp = -100;
4    }
5  }
```

ここが不具合の原因！

タイプミスして、100 の前にマイナス記号を付けちゃったのね。

そっかぁ。コンパイルエラーにはならないし、代入もできちゃうから気づかないよなぁ。

その翌日、今度は「冒険中にお城で会話をすると、なぜか勇者が理由もなく急死してゲームオーバーになる」という問題が見つかります。原因を調査したところ、またもやAさんが作ったリスト 10-3 の「王様クラス」に問題がありました。

リスト 10-3　「王様クラス」の問題点

King.java

```
1  public class King {
2    void talk(Hero h) {
3      System.out.println
           ("王様：ようこそ我が国へ、勇者" + h.name + "よ。");
4      System.out.println("王様：長旅疲れたであろう。");
5      System.out.println
           ("王様：まずは城下町を見てくるとよい。ではまた会おう。");
6      h.die();
7      ⋮
8    }
9  }
```

ここが不具合の原因。勇者が死ぬ！

先輩から「bye()を呼べ」と指示されたのを「die()」に聞き間違えたのかな？

うーん、英単語の意味を考えればわかりそうなものだけど…でも人間だから、間違えることもあるよね…。

今回の一連の不具合はAさんの不注意やスキル不足がきっかけで起こりました。しかし、見方を変えれば、簡単に「HPをマイナス100に設定できてしまうこと」や「王様が会話中に勇者を殺せてしまうこと」にも問題があります。

プログラムに以下のようなアクセス制御が盛り込まれていれば、このようなバグは事前に見つかったはずです。

Heroクラス以外からはhpフィールドに値を設定できない
die()メソッドを呼べるのはHeroクラスだけ

カプセル化はこのようなアクセス制御を実現し、想定外に発生する不具合を未然に防ぐためのしくみです。ぜひ次節以降のカプセル化の文法をしっかりマスターして、不具合が発生しにくいクラスを開発できるようになりましょう。

人間は必ずミスをする。だから不具合の原因を決して「人」に求めてはならない。原因は「ミスを未然に防ぐしくみがないこと」に求めるべきなんだ。

10.2 メンバに対するアクセス制御

10.2.1 4つのアクセス制御レベル

Java では、それぞれのメンバ(フィールドおよびメソッド)に対してアクセス制御の設定を行うことができます。ですが、それぞれのメンバに「A クラス、D クラス、R クラスからの利用は許す」「B クラス、Z クラスからの利用は許す」のように細かく指定すると、とても手間がかかってしまいます。そこで、表 10-1 のようにザックリと、4 段階からアクセス制御の方法を選ぶようなしくみになっています。

表 10-1　Java におけるアクセス制御の範囲と指定方法（メンバ編）

制限の範囲	名前	プログラム中の指定方法	アクセスを許可する範囲
制限が厳しい	private	private	自分自身のクラスのみ
↕	package private	（何も書かない）	自分と同じパッケージに属するクラス
	protected	protected	自分と同じパッケージに属するか、自分を継承した子クラス
制限が緩い	public	public	すべてのクラス

private や public などは**アクセス修飾子**(access modifier)と呼ばれ、フィールドやメソッドを宣言する際、先頭に記述することでアクセス制御が可能になります。

フィールドのアクセス制御

アクセス修飾子 フィールド宣言；

メソッドのアクセス制御

アクセス修飾子 メソッド宣言 {…}

現時点で表10-1の4つすべてを覚える必要はないよ。まずはpublicとprivateだけ覚えておけば十分だ。特にprotectedに関しては後の「継承」に関する章で学ぶから、それまで完全に忘れていても構わないよ。

10.2.2 privateを利用する

それではprivateによるアクセス制御を体験してみましょう。10.1.3項の「アクセス制御されない怖さ」では、Heroクラスのhpフィールドにマイナス100が設定されてしまいました。本来、HPフィールドはattack()したときに2ずつ減り、sleep()したときに回復すればよいので、他のクラスから変更できる必要はありません。よってHPはprivateにしておきましょう（リスト10-4）。

リスト10-4　HPをprivateにしたサンプルコード

Hero.java

```
1  public class Hero {
2    private int hp;
3    String name;
4    Sword sword;
5    static int money;
6    ⋮
7    void sleep() {
8      this.hp = 100;
9      System.out.println(this.name + "は、眠って回復した！");
10   }
11   ⋮
12 }
```

hpフィールドにprivateを指定したため、宿屋クラスのcheckIn()メソッドでは「hpフィールドへはアクセスできない」というコンパイルエラーが発生するようになります。

しかし、勇者のhpフィールドがいっさい変更できなくなるわけではない点に注意してください。privateなフィールドであっても、同じクラスのメソッドからであれば、リスト10-4のsleep()メソッドのように「this」を用いて読み書きすることができます。宿屋クラスのcheckIn()メソッドの中では、勇者のHPフィールドに直接100を代入できない代わりに、sleep()メソッドを呼び出すように修正すればよいのです。

privateアクセス修飾

privateであっても、自分のクラスからthis.~での読み書きは可能。

また、die()メソッドについても、王様など、ほかのクラスからみだりに呼び出されることがないようにprivateにします（リスト10-5）。

リスト10-5　die()メソッドをprivateとして指定する

Hero.java

```
1  public class Hero {
2    ⋮
3    private void die() {
4      System.out.println(this.name + "は死んでしまった！");
5      System.out.println("GAME OVERです。");
6    }
7    ⋮
8  }
```

これでdie()メソッドは外部のクラスからは呼び出せなくなりますが、同じクラス内にあるattack()メソッドからの呼び出し（リスト10-1の23行目）は問題ありません。

10.2.3 public や package private を利用する

勇者は戦うのが仕事ですから、いろいろなクラスから attack() メソッドが呼び出される可能性があります。よって attack() は、どのようなクラスからでも呼び出せるように public 指定を付けておきましょう（リスト 10-6）。

リスト 10-6　attack() メソッドは public として指定する

Hero.java

```
1  public class Hero {
2    ⋮
3    void sleep() {
4    ⋮
5    }
6    public void attack(Matango m) {
7    ⋮
8    }
9    ⋮
10 }
```


ちなみに、sleep() メソッドには public を付けないままにしています。この場合、package private を指定したと見なされ、同じパッケージに属するクラスからの呼び出しのみ可能になります。仮に図 10-4 のように Hero クラスが rpg.characters パッケージに属しているとすれば、他のパッケージに属する Slime クラスなどからは利用できなくなります。

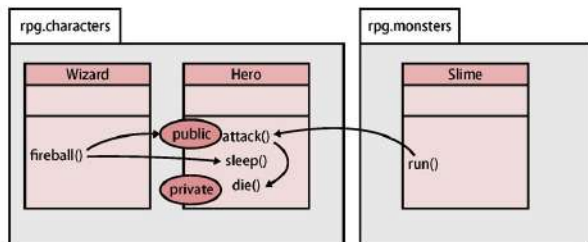



図 10-4　public、package private、private のアクセス制御

10.2.4 アクセス修飾の定石

private、public、package private についてはよくわかりました。
でも、どれを選べばよいのか迷っちゃいます…。

大丈夫、お決まりのパターンがあるんだよ。

どのメンバに、どのアクセス修飾子を指定すべきか、Java の文法では定められていません。アクセス修飾子は自由に指定できるので、「メンバの使われ方をプログラマがよく考慮した上で、最適なものを選ぶべき」というのが教科書的な答えです。しかし、「ほとんどのケースでは、このように指定しておけば大丈夫」あるいは「このような指定の仕方が基本」というパターン（定石）があります。

メンバに関するアクセス修飾の定石

・フィールドはすべて private
・メソッドはすべて public

とりあえずは、このパターンに沿ってアクセス修飾を行い、その後、Hero クラスの die() メソッドのように、クラス内部だけで利用するメソッドのみを private に指定し直すような「微調整」をすればよいのです。

クラスに対するアクセス修飾の定石

このあと 10.4 節ではクラスに対するアクセス修飾を学びますが、クラスは特に理由がない限り public で修飾するのが一般的です。

10.3 getter と setter

10.3.1 メソッドを経由したフィールド操作

前節の「定石」を見ていて思ったんですが、フィールドをすべて private にすると、外部からいっさい読み書きできなくなっちゃいませんか？

いや、そんなことはないよ。メソッドを経由すればフィールドにアクセスできるんだ。

ここでもう一度、リスト 10-4 をよく見てください。hp フィールドは private 指定され、ほかのクラスからはアクセスできなくなっています。しかし、外部のクラスから hp フィールドの値を変更できないかというと、そんなことはありません。

外部のクラスからであっても、attack() メソッドを呼べば HP を 2 減らすことができ、sleep() メソッドを呼べば HP を回復できます（図 10-5）。

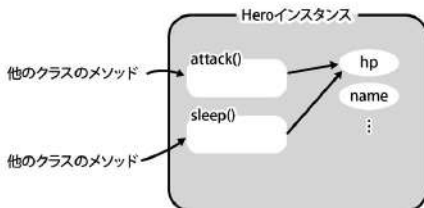



図 10-5　メソッドを経由しなければフィールドにはアクセスできない

hp フィールドの値を変化させるには、必ず attack() か sleep() を経由しなければならない点に注目してください。勇者の HP を増減するためには、このどちらかのメソッドを経由するほかありません。

つまり、他のクラス（宿屋クラスや王様クラス）の開発者がバグを含んだコード

を書いたとしても、勇者のHPをマイナス100に設定することは不可能です。

もし万が一、HPに異常な値が設定される不具合に直面しても、そのときはattack()かsleep()のどちらかのバグだということが簡単に予想できますから、不具合の修正もスムーズにできるでしょう。

基本的にフィールドはメソッド経由でアクセスするものなんですね。

10.3.2 単純にフィールド値を取り出すだけのメソッド

Heroクラスには名前を格納したnameというフィールドがあります。勇者の名前は、さまざまな場面で多くのクラスから利用されます。たとえば次のリスト10-7のように王様クラスの中でも利用されています。

リスト 10-7　王様クラスで利用される name フィールド

King.java

```
1  public class King {
2    void talk(Hero h) {
3      System.out.println
             ("ようこそ我が国へ、勇者" + h.name + "よ。");
4      ⋮
5    }
6  }
```

しかし、Heroクラスの全フィールドをprivateに設定すると、このKingクラスでは次のようなコンパイルエラーが発生してしまいます。

```
name は Hero で private アクセスされます。
```

Heroクラスのnameフィールドはprivateであるため、Kingクラスからはその存在が「見えない」のです。このままでは王様が勇者の名前を得ることができず、名前を呼ぶことができません。

そこで、Hero クラスにリスト 10-8 のような getName メソッドを追加して、王様が勇者の名前を知ることができるようにしましょう。

リスト 10-8　Hero クラスに getName メソッドを追加

Hero.java

```
1 public class Hero {
2   private String name;
3     :
4   public String getName() {
5     return this.name;
6   }
7 }
```

そして King クラスでは、name フィールドにアクセスしている部分を、getName() を呼び出すように修正すれば完成です（リスト 10-9）。

リスト 10-9　King クラスの talk() メソッド内を以下のように修正

King.java

```
1 public class King {
2   void talk(Hero h) {
3     System.out.println
          ("王様：ようこそ我が国へ、勇者" + h.getName() + "よ。");
4       :
5   }
6 }
```

10
章

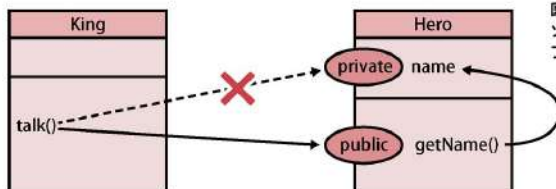



図 10-6　getName() メソッドを経由して name フィールドにアクセスする

このgetName()メソッドは、attack()メソッドなどとは異なり、単にnameフィールドの中身を呼び出し元に返すだけの単純なメソッドです。このようなメソッドを総称してgetter（ゲッター）メソッドといいます。

10.3.3 getterメソッドの書き方

ある特定のフィールド値を単に取り出すだけのメソッドは、すべてgetterメソッドと言えます。このgetterメソッドの書き方にも「定石」があります。

getterメソッドの定石

```
public　値を取り出すフィールドの型　getフィールド名()　{
　return this.フィールド名;
}
```

メソッド名の最初の3文字を「get」にし、それに続けて「フィールド名の先頭を大文字にしたもの」にします。 たとえば、フィールド名がnameならgetName()となります（例外として戻り値がboolean型の場合のみisXxxx()というメソッド名にすることがあります）。このメソッド名の付け方はJava開発者の間で常識になっている風習みたいなものだと思ってください。

たとえば、開発現場で「nameのgetter」などの言葉が飛び交うことがある。これはnameフィールドに対するgetterメソッド、つまりgetName()メソッドのことを指しているんだ。

10.3.4 単純にフィールドに値を代入するだけのメソッド

getterメソッドとは逆に、ある特定のフィールドに指定された値を単に代入するだけのメソッドをsetter（セッター）メソッドといいます。setterメソッドも、その記述方法には定石があります。

setter メソッドの定石

public　void　set フィールド名 (フィールドの型　任意の変数名)　{
　this. フィールド = 任意の変数名 ;
}

たとえば、Hero クラスについて、name フィールドに対応する setter メソッドを追加するとリスト 10-10 のようになります。

リスト 10-10　setter メソッドの例

Hero.java

```
1  public class Hero {
2    ⋮
3    public void setName(String name) {
4      this.name = name;
5    }
6  }
```


カプセル化とは関係ないことだが、このコードの代入式では左辺に this. を忘れると大事故につながることを再認識しておこう。なぜかわからない人は 8.2.5 項を読み返してほしい。

10.3.5 getter／setter の存在価値

ちょっと待ってください！ せっかく name フィールドを private にして外部のアクセスから守ったのに、また getter／setter を用意してアクセスを外部に開放したら private の意味がないんじゃありませんか？

いや、そんなことはないよ。getter／setterには重要な存在価値があるんだ。

Heroクラスのnameフィールドに関係する部分だけを取り出して、カプセル化の前（リスト10-11）と、カプセル化の後（リスト10-12）のコードを見比べてみましょう。

リスト10-11　カプセル化を行う前

Hero.java

```
1 public class Hero {
2   ⋮
3   String name;
4   ⋮
5 }
```

リスト10-12　カプセル化を行った後

Hero.java

```
1 public class Hero {
2   private String name;
3   public String getName() {
4     return this.name;
5   }
6   public void setName(String name) {
7     this.name = name;
8   }
9 }
```

どちらもnameフィールドの読み書きができることに違いはありません。むしろコードの行数が増えるため、getterやsetterを利用することに意味が感じられないかもしれません。しかし、getterとsetterには次のようなメリットがあります。

メリット1：Read Only、Write Onlyのフィールドを実現できる

リスト10-12のsetName()メソッドを削除すれば、nameフィールドを「外部から読めるが書き換えられない（Read Only）」フィールドにできます。実際のプログラミングでも「外部から自由に読めるようにしたいが、変更されては困る」というフィールドが必要になることがありますが、その際に多用されるテクニックです。

また、あまり使われませんが、setterメソッドだけを準備して「外部から自由に書き換えできるが、読めない（Write Only）」フィールドも作成可能です。

メリット2：フィールドの名前など、クラスの内部設計を自由に変更できる

たとえば将来、何らかの理由でnameというフィールド名をnewNameに変更したくなったとしましょう。もしgetter／setterを準備せず、他のクラスから直接、nameフィールドを読み書きしていた場合、他のクラスのすべての開発者に「アクセスするフィールド名を変更してもらうお願い」をして回らなければなりませ

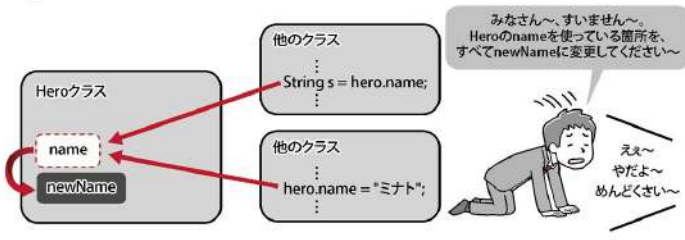
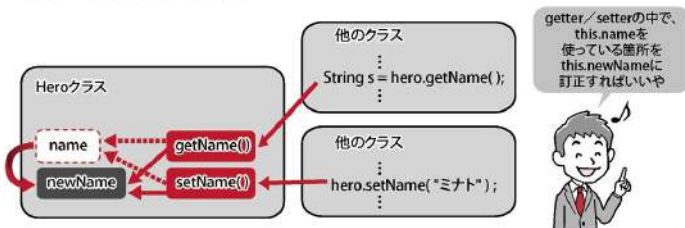



図10-7　外部から隠せば、内部仕様の変更が柔軟に行える

ん（図 10-7 の上側）。

一方、name フィールドを隠し、外部からは getter／setter 経由で読み書きさせるのならフィールド名の変更は自由です。なぜなら getter や setter の内部でフィールドを使っている箇所だけを修正すればよく、getName() や setName() を呼び出している他の開発者には影響がないからです（図 10-7 の下側）。

メリット 3：フィールドへのアクセスを検査できる

クラス外部から name フィールドの値を書き換えたい場合、setter を使う必要が生じます。つまり、「setter を実行せずに、name フィールドを書き換えることは不可能」です。

このことを利用して、setter で「設定されようとしている値が妥当かを検査する」ことも Java プログラミングの定石です。たとえばリスト 10-13 のように setName() を改良してみましょう。

リスト 10-13　setter メソッドの中で値の妥当性をチェックする

```
1  private String name;
2  public void setName(String name) {
3    if (name == null) {
4      throw new IllegalArgumentException
         ("名前がnullである。処理を中断。");
5    }
6    if (name.length() <= 1) {
7      throw new IllegalArgumentException
         ("名前が短すぎる。処理を中断。");
8    }
9    if (name.length() >= 8) {
10     throw new IllegalArgumentException
         ("名前が長すぎる。処理を中断。");
11   }
12   this.name = name;
13 }
```

「throw new IllegalArgumentException」は、今の段階では「エラーを出してプログラムが強制停止する命令」と理解しておいてほしい。

このsetName()メソッドは、nameフィールドの値を変更しようとするたびに検査を行います。もし、リスト10-14のような問題のあるプログラムを実行すると、プログラムはきちんと停止するため、開発者はバグに気づくことができます。

リスト10-14　setNameが正しく機能するかを確認する

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h = new Hero();
4      h.setName("");
5    }
6  }
```

長さ0文字の名前をセットしようとする

実行結果

```
Exception in thread "main" java.lang.IllegalArgumentException:
名前が短すぎる。処理を中断。
  at Hero.setName(Hero.java:17)
  at Main.main(Main.java:4)
```


ぜひ、「どのように誤っても、どのような悪意を持ったアクセスでも、外部から絶対に不正な値を設定できない」安全・安心なクラスのプログラミングを目指して、検査を徹底させた強固なsetterを書くように心がけてください。

10.4 クラスに対するアクセス制御

10.4.1 2つのアクセス制御レベル

メンバへのアクセス制御と同じく、あるクラス全体に対してアクセス制御を設定することができます。クラスのアクセス制御レベルは、表10-2のように2種類しかありません。

表10-2　クラスへのアクセス制御の指定方法と範囲

名前	Javaでの記述	許可する範囲	制限
package private	（何も書かない）	自分と同じパッケージに属するクラス	厳しい
public	public	すべてのクラス	緩い

これまで、クラス宣言の前にはpublicを付けると丸暗記していましたが、実はクラス宣言の先頭にpublicという記述がない場合、そのクラスは同一パッケージに属するクラスからのアクセスのみ許可されます。

他のパッケージに属するクラスからのアクセスが禁止されるわけですが、イメージとしては「他のパッケージに属するクラスから、そのクラスの存在自体が見えなくなる」と捉えたほうがよいでしょう。

そのため、たとえpublic指定されたメソッドであっても、属するクラスが

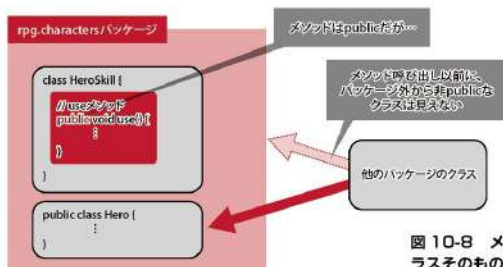



図10-8　メソッドを呼び出す以前にクラスそのものが見えない

package private ならば、別パッケージのクラスからはそのメソッドを呼び出せなくなります（図 10-8）。

10.4.2 非 public クラスとソースファイル

別パッケージのクラスから見えなくなってしまう package private クラスですが、その代わりに public クラスでは許可されない次の 2 つが許されています。

非 public クラスの特徴

①クラスの名前はソースファイル名と異なってもよい。
②1 つのソースファイルに複数宣言してもよい 。

これまでは「1 つのファイルに 1 つのクラス」「ファイル名＝クラス名」が原則だと紹介してきましたが、より正確には、「1 つのファイルに 1 つの public クラス」「ファイル名＝ public クラス名」というルールです（図 10-9）。

public が付いていないクラスは、どのソースファイルにいくつ宣言されても構いません。なお、ソースファイルに public クラスが 1 つも含まれない場合、ソースファイル名は自由に決めることができます。

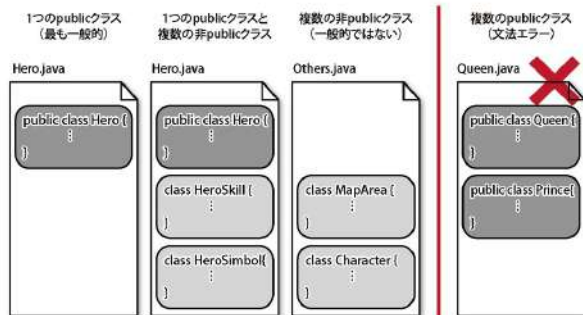



図 10-9　1 つのファイルに複数のクラスを宣言するバリエーション

10.5 カプセル化を支えている考え方

10.5.1 メソッドでフィールドを保護する

この章では public や private を用いて、クラスやメンバに対するアクセス制御の方法を学びました。

特に重要なメンバのアクセス制御では、特別な理由がない限り、「フィールドは private として外部から隠し、必ず setter／getter メソッド経由でアクセスする」という定石についても理解できたと思います。フィールドはメソッドによって守られており、外部から直接アクセスできないようにするのです（図 10-10）。

身分の高い王様は護衛に守られていて、直接は謁見できないのと似ていますね。

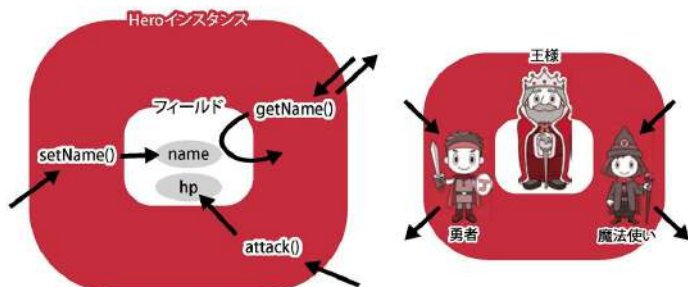



図 10-10　王様（フィールド）は勇者と魔法使い（メソッド）に守られており、外部から直接アクセスできない

この図のように、「外部から直接触られないよう、メソッドという殻（カプセル）によってフィールドが保護されている」ように見えることから、カプセル化という名前が付いています。

ところで、なぜカプセル化ではメソッドではなくフィールドを保護しようとするのでしょうか？ それは、メソッドよりフィールドのほうが異常な状態（不具合）になりやすいからです。

メソッドの処理内容は、プログラミング段階で決定し、一度コンパイルされればプログラム実行中に変化することはありません。一方、フィールドの値は、プログラムが動作する間に刻々と変化していきます。そのため、動作中に異常な値になる危険性もありえます。結果的に、不具合の多くは「フィールドに予期しない値が入る」という形で発現します。

したがって、「プログラムの不具合を減らすためには、メソッドよりもフィールドを保護することが重要」なのです。プログラムの不具合を防止するために、どんどんカプセル化を活用していきましょう。

適切にカプセル化されていれば、インスタンスは大切なフィールドを直接外部にさらすことなく、互いに公開した setter／getter やその他のメソッドを呼び合うことで、安全に連携できるのです。

10.5.2 カプセル化の本質

よし、これでカプセル化もマスターだ！ でもやっぱり、「オブジェクト指向といえば継承！」なんですよね？

確かに次章で学ぶ継承のほうが有名だけど、カプセル化こそがオブジェクト指向の本質を支えているんだ。

第７章で学んだオブジェクト指向の本質を思い出してください。システムやプログラムというのは、突き詰めれば「現実世界における何かの相互作用」を自動化するためのものでした。そして「現実世界の登場人物たちの動きを、そっくり仮想世界に再現する」ことがオブジェクト指向の基本的な考え方です。

では、このオブジェクト指向の世界において、「バグ」「不具合」とはいったい何でしょうか？ それは、すなわち 次の状態にほかなりません。

不具合とは

「そもそもバグとは、現実世界と仮想世界が食い違ってしまうこと」

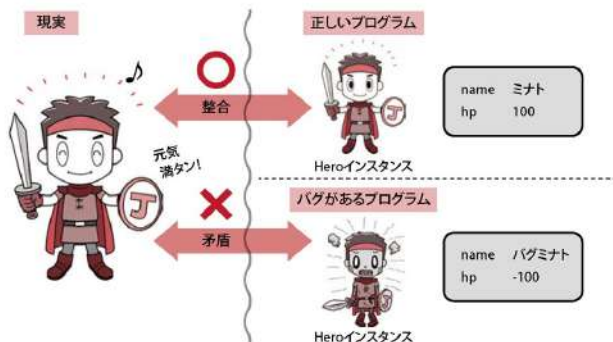



図 10-11　「不具合」とは、現実と矛盾した状態の仮想世界のこと

「実際の勇者は元気なのに、ゲームではなぜか HP が -100 になっている（図 10-11）」「実際の在庫数は 800 なのに、システム内ではなぜか 80 になってしまう」からバグなのです。

しかし、この章で学んだカプセル化を使えば、どのように利用されても、フィールドに不正な値が入ってしまうことのない、「現実の登場人物と矛盾しないクラス」を作ることができます。 そして、その「現実の登場人物と矛盾しないクラス」を集めてプログラムを作れば、「現実世界と矛盾しないプログラム」になるという考えがカプセル化の本当の狙いなのです。

以降の章で学ぶ「継承」や「多態性」に比べれば、この章で学んだカプセル化は比較的簡単で、文法や効果にも華やかさはありません。しかし「現実世界を忠実にまねる」というオブジェクト指向の本質と直結している、最も重要な位置づけにある機能の 1 つだといえるでしょう。

10.6 第10章のまとめ

この章では、次のようなことを学びました。

カプセル化の概要

- カプセル化を用いるとメンバやクラスについてアクセス制御が可能になる。
- 特に、フィールドに「現実世界ではありえない値」が入らないように制御する。

メンバに対するアクセス修飾

- private 指定されたメンバは、同一クラス内からしかアクセスできない。
- package private 指定されたメンバは、同一パッケージ内のクラスからしかアクセスできない。なお、メンバ宣言に特定のアクセス修飾子を付けなければ package private になる。
- public 指定されたメンバは、すべてのクラスからアクセスできる。

クラスに対するアクセス修飾

- package private 指定（修飾子なし）で宣言されたクラスは、同一パッケージ内のクラスからしかアクセスできない。
- public 指定されたクラスは、すべてのクラスからアクセスできる。

カプセル化の定石

- クラスは public、メソッドは public、フィールドは private で修飾する。
- フィールドにアクセスするためのメソッドとして getter や setter を準備する。
- setter 内部では引数の妥当性検査を行う。

練習問題

練習 10-1

次の２つのクラス「Wizard（魔法使い）」「Wand（杖）」のすべてのフィールドとメソッドについて、カプセル化の定石に従ってアクセス修飾子を追加してください（Wizard クラスにコンパイルエラーが発生しますが、それは構いません）。

Wand.java
```
1  public class Wand {
2    String name;     // 杖の名前
3    double power;    // 杖の魔力
4  }
```

Wizard.java
```
1  public class Wizard {
2    int hp;
3    int mp;
4    String name;
5    Wand wand;
6    void heal(Hero h) {
7      int basePoint = 10;                       // 基本回復ポイント
8      int recovPoint = (int) (basePoint * this.wand.power);
9                                                // 杖による増幅
10     h.setHp(h.getHp() + recovPoint);          // 勇者のHPを回復させる
11     System.out.println
           (h.getName() + "のHPを" + recovPoint + "回復した!");
12   }
13 }
```

練習 10-2

問題 10-1 で作成した Wand クラスと Wizard クラスのすべてのフィールドについて、定石に従って getter メソッドと setter メソッドを作成してください。また、Wizard クラスの heal メソッドで発生しているコンパイルエラーを解決してください。ただし、setter メソッドに関しては引数の妥当性検証は不要です。

練習 10-3

問題 10-2 で作成した Wand クラスと Wizard クラスの各 setter メソッドについて、以下 4 種類のルールに従って引数の妥当性検証を追加してください。不正な値がセットされそうになった場合には、「throw new IllegalArgumentException(" エラーメッセージ ");」を記述してプログラムを中断させてください。

①魔法使いや杖の名前は null であってはならず、必ず 3 文字以上である。
②杖の魔力による増幅率は、0.5 以上 100.0 以下である。
③魔法使いの杖は null であってはならない。
④魔法使いの HP と MP は 0 以上である。ただし HP については負の値が設定されそうになると代わりに 0 が設定される。

練習問題の解答

問題 10-1 の解答

Wand.java

```
1  public class Wand {
2    private String name;      // 杖の名前
3    private double power;     // 杖の魔力
4  }
```

Wizard.java

```
1  public class Wizard {
2    private int hp;
3    private int mp;
4    private String name;
5    private Wand wand;
6    public void heal(Hero h) {
7      int basePoint = 10;                 // 基本回復ポイント
8      int recovPoint = (int) (basePoint * this.wand.power);
9                                          // 杖による増幅
10     h.setHp(h.getHp() + recovPoint);    // 勇者のHPを回復
11     System.out.println
         (h.getName() + "のHPを" + recovPoint + "回復した!");
12   }
13 }
```

問題 10-2 の解答

Wand.java

```
1  public class Wand {
2    private String name;      // 杖の名前
3    private double power;     // 杖の魔力
```

```
4    public String getName() { return this.name; }
5    public void setName(String name) { this.name = name; }
6    public double getPower() { return this.power; }
7    public void setPower(double power) { this.power = power; }
8  }
```

Wizard.java

```
1  public class Wizard {
2    private int hp;
3    private int mp;
4    private String name;
5    private Wand wand;
6    public void heal(Hero h) {
7      int basePoint = 10;                  // 基本回復ポイント
8      int recovPoint =                     // 杖による増幅
           (int) (basePoint * this.getWand().getPower());
9      h.setHp(h.getHp() + recovPoint);     // 勇者のHPを回復
10     System.out.println
           (h.getName() + "のHPを" + recovPoint + "回復した!");
11   }
12   public int getHp() { return this.hp; }
13   public void setHp(int hp) { this.hp = hp; }
14   public int getMp() { return this.mp; }
15   public void setMp(int mp) { this.mp = mp; }
16   public String getName() { return this.name; }
17   public void setName(String name) {this.name = name; }
18   public Wand getWand() { return this.wand; }
19   public void setWand(Wand wand) { this.wand = wand; }
20 }
```

問題10-3の解答

Wand.java

```
1  public class Wand {
2    private String name;      // 杖の名前
3    private double power;     // 杖の魔力
4    public String getName() { return this.name; }
5    public void setName(String name) {
6      if (name == null || name.length() < 3 ) {
7        throw new IllegalArgumentException
           ("杖に設定されようとしている名前が異常です");
8      }
9      this.name = name;
10   }
11   public double getPower() { return this.power; }
12   public void setPower(double power) {
13     if (power < 0.5 || power > 100.0) {
14       throw new IllegalArgumentException
           ("杖に設定されようとしている魔力が異常です");
15     }
16     this.power = power;
17   }
18 }
```

Wizard.java

```
1  public class Wizard {
2    private int hp;        private int mp;
3    private String name;  private Wand wand;
4    public void heal(Hero h) {
5      int basePoint = 10;                   // 基本回復ポイント
6      int recovPoint =                      // 杖による増幅
         (int) (basePoint * this.getWand().getPower());
7      h.setHp(h.getHp() + recovPoint);     // 勇者のHPを回復
8      System.out.println
```

```
          (h.getName() + "のHPを" + recovPoint + "回復した!");
 9    }
10    public int getHp() { return this.hp; }
11    public void setHp(int hp) {
12      if (hp < 0) { this.hp = 0; } else { this.hp = hp; }
13    }
14    public int getMp() { return this.mp; }
15    public void setMp(int mp) {
16      if (mp < 0) {
17        throw new IllegalArgumentException
            ("設定されようとしているMPが異常です");
18      }
19      this.mp = mp;
20    }
21    public String getName() { return this.name; }
22    public void setName(String name) {
23      if (name == null || name.length() < 3) {
24        throw new IllegalArgumentException
            ( "魔法使いに設定されようとしている名前が異常です" );
25      }
26      this.name = name;
27    }
28    public Wand getWand() { return this.wand; }
29    public void setWand(Wand wand) {
30      if (wand == null) {
31        throw new IllegalArgumentException
             ( "設定されようとしている杖がnullです" );
32      }
33      this.wand = wand;
34    }
35  }
```

第11章

継承

たくさんのクラスを作るうちに、
「以前作ったクラスにとてもよく似ているが、一部だけ違うクラス」
を作る必要に迫られ、めんどうに感じることも増えてきます。
この章では、このような課題を解決してくれるオブジェクト指向の
花形機能、「継承」の基本を学びましょう。

CONTENTS

11.1 継承の基礎

11.1.1 似かよったクラスの開発

Java で大きなプログラムを書き始めると、**以前作ったクラスと似かよったクラスを作る**必要に迫られることがあります。

「ほとんど同じだけどフィールドが 2 つほど多い」あるいは「ほとんど同じだけどメソッドが 1 つ多い」という具合です。そのようなクラスは、どうすれば効率よく作れるでしょうか。

例として勇者クラス（Hero クラス）を取り上げて考えてみましょう。この章では理解しやすさのために、リスト 11-1 のような単純な Hero クラスから始めます。

リスト 11-1 「戦う」と「逃げる」しかできない Hero クラス

Hero.java

```
1  public class Hero {
2    private String name = "ミナト";
3    private int hp = 100;
4    // 戦う
5    public void attack(Matango m) {
6      System.out.println(this.name + "の攻撃！");
7      m.hp -= 5;
8      System.out.println("5ポイントのダメージをあたえた！");
9    }
10   // 逃げる
11   public void run() {
12     System.out.println(this.name + "は逃げ出した！");
13   }
14 }
```

このHeroは冒険するにつれ進化してゆき、以下のような能力を持ったSuperHeroという職業になれるとしましょう。

スーパーヒーローはfly()で空を飛ぶことができ、land()で着地できる。
ヒーローができるすべての動作は、スーパーヒーローもできる。

では、リスト11-1のHeroクラスを元に、SuperHeroクラスを開発してみましょう。

簡単よ。Heroのコードをコピー＆ペーストし、クラス名をSuperHeroに変更して、それにfly()とland()のメソッドを足せば、あっという間にできあがり！ 名付けて「コピペ解決法」よ！

朝香さんはHeroクラスを元に、次のようなリスト11-2のコードを書きました。うまく動作するでしょうか？

リスト11-2　朝香さんが作成したSuperHeroクラス

SuperHero.java

```
1  public class SuperHero {          ← クラス名を書き換えた
2    private String name = "ミナト";
3    private int hp = 100;
4    private boolean flying;         ← flyingフィールドを追加
5    // 戦う
6    public void attack(Matango m) {
7      System.out.println(this.name + "の攻撃！");
8      m.hp -= 5;
9      System.out.println("5ポイントのダメージをあたえた！");
10   }
11   // 逃げる
12   public void run() {
13     System.out.println(this.getName() + "は逃げ出した！");
14   }
```

```
15    // 飛ぶ
16    public void fly() {
17      this.flying = true;
18      System.out.println("飛び上がった！");
19    }
20    // 着地する
21    public void land() {
22      this.flying = false;
23      System.out.println("着地した！");
24    }
25  }
```

fly() を追加

land() を追加

11.1.2 「コピペ解決法」の問題点

さすが朝香くん。あっという間に SuperHero を作りあげたね。

はいっ。大学のレポート作成の課題では、ネットで調べてコピペ使いまくりましたから！

レポートの話は聞かなかったことにするけど、この方法だと後で困るかもしれないよ。

朝香さんのように元となるコードをコピー&ペーストして、それに新しい機能を追加すれば、簡単に元のクラスを発展させることができます。解決方法としてはとてもシンプルですし、コードとしても問題なく動作するでしょう。

しかし、この方法によって作成された SuperHero クラスには、次のような 2 つの問題があります。

■追加・修正に手間がかかる

Hero クラスに新しいメソッドを加えたときや、Hero クラス内のメソッドを変更した場合、その変更を SuperHero クラスにも行う必要があります。なぜ

ならスーパーヒーローとは、「たくさんいるヒーローの中でも特に優れたひとにぎりの者」だからです。SuperHero は Hero ができることは当然すべてできなければなりません。

■把握や管理が難しくなる

SuperHero クラスは Hero クラスを元にしているので、この2つのクラスでソースコードの大半が重複することになります。これによりプログラム全体の見通しが悪くなり、メンテナンスがしづらくなります。

もしかしたら今後、Hero を元にした別のクラス(たとえば SuperHero や HyperHero、LegendHero、MagicalHero など)を作る必要が出てくるかもしれません。すると Hero クラスに何か変更があるたびに、すべての〜 Hero クラスに対して Hero クラスと同じ修正を行う必要が生じます。これは、とてもめんどうですね。

11.1.3 継承による解決

このゲームは後で職業を増やしていきたいんです。ですから新しい職業クラスを作るたびにコピペしていると、後から問題が出て困りそうです。

そうだね。そもそも同じコードが何か所にも分散して書かれていることが諸悪の根源だ。

「コピペ解決法」を用いて類似したクラスを作成していくと、将来、元となったクラスが変更された際に、すべての類似クラスも修正しなければなりません。

しかし Java には、このようなことを懸念することなく類似したクラスを作ることを可能にする機能「**継承**」があります。これを使えば、SuperHero クラスは次ページのリスト 11-3 のようにスッキリと記述することができます。

リスト 11-3　Hero クラスを継承して SuperHero を作成する

SuperHero.java

```
1  public class SuperHero extends Hero {
2    private boolean flying;
3    public void fly() {
4      this.flying = true;
5      System.out.println("飛び上がった！");
6    }
7    public void land() {
8      this.flying = false;
9      System.out.println("着地した！");
10   }
11 }
```

「基本的には Hero と同じ」と宣言

追加した flying

追加した fly()

追加した land()

ポイントは、1 行目の extends です。この修飾子を用いた「class SuperHero extends Hero」という宣言は、「基本的に Hero クラスをベースにして SuperHero クラスを定義するので、Hero と同じメンバの定義は省略します（違いだけを記述します）」という意味になります。

Hero クラスを継承しているから「新しく増えたメンバだけ」を SuperHero クラスに書けばいいわけですね。

継承を用いたクラスの定義

```
class クラス名 extends 元となるクラス名 {
  親クラスとの「差分」メンバ
}
```

この SuperHero クラスがインスタンス化されるときに、JVM は「省略されているけれども、SuperHero クラスは Hero クラスに含まれている run()、

attack()、hp、name も持っている」と判断してくれます。

図 11-1　2 つのクラス定義に基づき、1 つのインスタンスが生成される

よって、**SuperHero クラスのソースコードには run() メソッドがありませんが、インスタンス化されれば run() メソッドを呼び出せます**（リスト 11-4）。

リスト 11-4

Main.java

```
public class Main {
  public static void main(String[] args) {
    SuperHero sh = new SuperHero();
    sh.run();
  }
}
```

このように、extends 修飾子を用いることによって、**元となるクラスの「差分」だけを記述して新たなクラスを宣言**することができます。

新たに定義するクラス (SuperHero) に着目すると、まるで「元となるクラス (Hero) から、メンバが自動的に引き継がれているように見える」ことから「継承」という名前が付いています。

もし Hero クラスに将来メソッドやフィールドの宣言が追加されれば、SuperHero でも自動的に使えるようになるのね。

そのとおり。コピペ解決法よりもエレガントな方法だろう？

11.1.4 継承関係の表現方法

今回の例では、Heroクラスを継承してSuperHeroクラスを作りました。このような2つのクラスの関係を継承関係といい、その元となるクラスを「スーパークラス」「基底クラス」「親クラス」などと呼び、新たに定義されるクラスを「サブクラス」「派生クラス」「子クラス」などと呼びます。

図11-2　図における継承関係の記述方法

なお、継承関係を図で表現する場合は、図11-2のような矢印で記述するルールになっています。

先輩、この図の矢印ですけど、方向が違うんじゃありませんか？
Heroクラスをベースに SuperHeroクラスを作るんですから、
下向きの矢印だと思うんですけど…。

いや、図の描き方としてはこれで正しいんだ。

慣れるまではこの矢印の方向に違和感を覚えるかもしれません。この矢印を直感とは逆に描くのには理由がありますが、それは本章の最後に説明します。今のところは、「クラス図では、継承の矢印は直感と逆で、子クラスから親クラスに向かって引く」とだけ覚えておいてください。

11.1.5 継承のバリエーション

継承のバリエーションは、2つのクラスの関係にはとどまりません。1つのクラスをベースとして、複数の子クラスを定義することもできますし、孫クラスや曾孫クラスを定義することもできます（図11-3）。

ただし、Javaでは許されていない継承の構図が1つだけあります。

図11-4のように、複数のクラスを親として1つの子クラスを定義することを**多重継承**といいますが、Javaではこれを許可していません。

図11-3 複数の子クラス、孫クラス

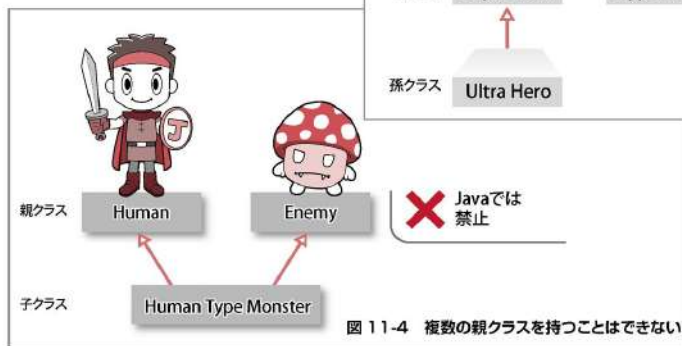



図11-4 複数の親クラスを持つことはできない

11.1.6 オーバーライド

SuperHeroのrun()を呼んだときに表示される「逃げ出した」という表示を「撤退した」に変えたいけど、どうしよう…。

何やってるのよ、SuperHeroクラスのrun()メソッドを書き換えればいいじゃない！ …って、SuperHeroクラスにはrun()メソッドの宣言がないんだっけ！

リスト 11-3(p.412)のコードを再度確認してください。SuperHero クラスには、fly() と land() の 2 つのメソッドしか定義されていません。SuperHero クラスの run() メソッドの動きだけを変えたい場合、どうすればよいでしょうか？

このような場合には、SuperHero クラスのコードに新しい run() メソッドを記述することができます。親クラスである Hero にも run() メソッドはありますが、子クラス SuperHero でも再度 run() メソッドを定義するのです。

リスト 11-5

SuperHero.java

```
public class SuperHero extends Hero {
  private boolean flying;
  public void fly() {
    ：
  }
  public void land() {
    ：
  }
  public void run() {
    System.out.println("撤退した");
  }
}
```


リスト 11-6

Main.java

```
public class Main {
  public static void main(String[] args) {
    Hero h = new Hero();
    h.run();
    SuperHero sh = new SuperHero();
    sh.run();
  }
}
```

実行結果
ミナトは逃げ出した
撤退した

リスト 11-5 の 1 行目で SuperHero クラスを定義する際、「基本的に Hero クラスをベースにする」と宣言したものの、9 行目で改めて run() メソッドを違う内容で定義し直しています（内容を上書きしている）。このように、親クラスを継承して子クラスを宣言する際に、親クラスのメンバを子クラス側で上書きすることを、**オーバーライド**（override）といいます。

以前に学んだオーバーロード（5.4 節）と**名前が似ているが、まったく異なるもの**なので混同しないでほしい。

継承を用いて子クラスに宣言されたメンバ

①親クラスに同じメンバがなければ、そのメンバは「追加」になる。
②親クラスに同じメンバがあれば、そのメンバは「上書き変更」される。

11.1.7 継承やオーバーライドの禁止

文字数の長さ制限がある LimitString クラスを作りたいんですけど、継承がうまくできなくて。

なるほど、String クラスの継承に失敗しているようだね。

朝香さんが作成しようとしているクラスは次のとおりです。

LimitString.java

```
public class LimitString extends String {
  ⋮
}
```

String クラスを継承

しかし、このソースコードをコンパイルしようとするとエラーが発生します。なぜなら、実は継承しようとしている String クラス（java.lang.String）は、「このクラスを継承して他のクラスを作ってはいけません（継承禁止）」と特別に指定されているクラスだからです。

Java の API リファレンスを見ると、String クラスは次のように宣言されていることがわかります。

```
public final class String extends Object…
```

Java では、**宣言時に final が付けられているクラスは継承できない**ことになっています。

もちろん、私たちがクラスを作成する際にも final を付ければ「継承禁止」にできます。たとえば、Main クラスの継承を禁止にするには、クラス宣言に final を追加します。

Main.java

```
1  public final class Main {
2    public static void main(String[] args) {
3      // メインメソッド
4    }
5  }
```

そもそもなぜ String クラスは継承禁止クラスとして宣言されているんですか？ 継承できたほうが絶対便利なのに。

String クラスを作った人は「String クラスのおかしな類似品」が世の中に出回る危険性を考えたのだろうね。

不具合なく完璧に動作するクラスがあっても、技術力がない人がそのクラスを継承し、オーバーライドによってメソッドの内容をメチャクチャに上書きしてしまったら、「異常な動作をする子クラス」ができてしまいます。

この「異常な子クラス」は、親クラスと似ているようでその内容はまったく異なる困った類似品であり、バグの原因となる危険性があります。特にStringクラスはプログラム内で多用される大切なクラスなので、「正しく動作しないStringの類似品」が出回ると致命的な不具合の原因になりかねません。

このような理由からStringクラスにはfinalが付けられていて、すべてのメソッドはオーバーライドできないようになっています。

確かに、Stringはどんなプログラムでもほぼ確実に使うものだし、バグがあったら大変なことになっちゃいます。

もし、クラスの継承は許可するものの、一部のメソッドについてのみオーバーライドを禁止したい場合は、そのメソッドの宣言にfinalを付けてください。**宣言にfinalが付けられたメソッドは、子クラスでオーバーライドができない**ことになっています。

リスト 11-7

Hero.java

```
1  public class Hero {
2    ⋮
3    public final void slip() {
4      this.hp -= 5;
5      System.out.println(this.getName() + "は転んだ！");
6      System.out.println("5のダメージ");
7    }
8    public void run() {
9      System.out.println(this.getName() + "は逃げ出した！");
10   }
11   ⋮
12 }
```

finalが付いているslip()メソッドは子クラスでオーバーライド禁止

runメソッドは子クラスでオーバーライド可能

11章

継承やオーバーライドの禁止

・クラス宣言に final を付けると、継承禁止
・メソッド宣言に final を付けると、オーバーライド禁止

実は継承の基本的な使い方や知識は、ここまでに紹介した内容がすべてだよ。

なんだぁ…花形機能というから、どんなに難しいかビクビクしていました。

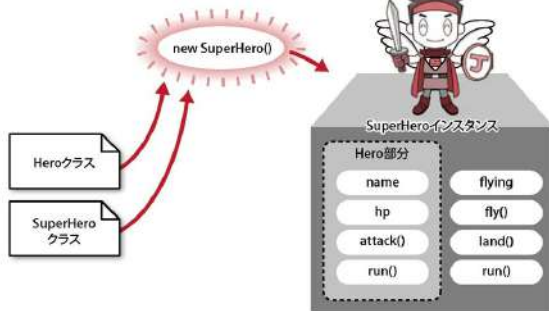
とりあえずここまでの知識で大丈夫だ。どんどん継承を使ってほしい。すぐにいくつかの不自由に直面し、より深く継承を理解する必要が出てくるはずだからね。

11.2 インスタンスの姿

11.2.1 インスタンスの多重構造

より踏み込んで継承を使いこなすには、継承を用いて定義されたSuperHeroのようなクラスから生まれたインスタンスが、実際にどのような姿をしていて、どのようにふるまうかを理解しておくことがとても重要です。継承を用いているインスタンスの姿を正しくイメージできれば、次節以降はもちろん、第12章の「高度な継承」や第13章の「多態性」もスムーズに理解できるでしょう。

では、SuperHeroのインスタンスの姿をイメージ図で見てみましょう。

図 11-5　SuperHeroインスタンスのイメージ図

このインスタンスは、外から見れば1つのSuperHeroインスタンスなのですが、内部にHeroクラスから生まれたHeroインスタンスを含んでおり、全体として**二重構造になっている**ことに着目してください。

スーパーヒーローさんは、胸の中に「普通のヒーローとしての自分」を秘めているのね。

これ以降では、外側の部分を「子インスタンス部分」、内側の部分を「親インスタンス部分」と呼び、このイメージ図を通してさまざまな呼び出しや動作原理を考えていきましょう。

ちなみに、「親・子・孫」と3つのクラスが継承関係にある場合、孫クラスのインスタンスは三重構造になるよ。

11.2.2 メソッドの呼び出し

インスタンスの外部からメソッドの実行依頼が届く（呼び出しがある）と、多重構造のインスタンスは、**極力、外側にある子インスタンス部分のメソッドで対応**しようとします。

たとえば、fly() が呼び出されれば SuperHero クラスで定義された fly() メソッドが動きます。一方、attack() への呼び出しは、まずは外側の子インスタンス部分で対応しようとしますが、外側に attack() は存在しません。そこで内側の親インスタンス部分の attack() メソッドに呼び出しが届き、それが動作します。

run() メソッドは、SuperHero と Hero の両方のクラスで定義（オーバーライド）されており、SuperHero インスタンスは内部に**「SuperHero としての逃げ方」と「Hero としての逃げ方」の両方を持っています**。しかし、外部から run() を呼び出された場合、外側にある SuperHero としての run() が優先的に動作するため、内側の run() が動くことはありません。

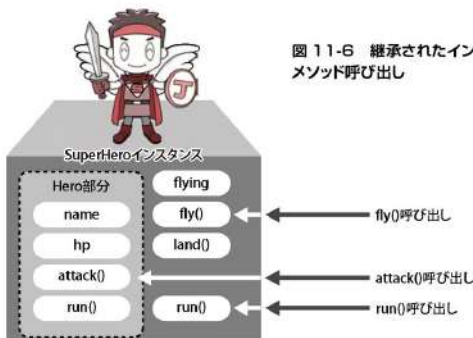



図 11-6　継承されたインスタンスのメソッド呼び出し

子クラスでrun()を定義しても、親クラスのrun()メソッドは上書きされてなくなるわけではないんですね。

親クラスのrun()も子クラスのrun()も両方ともインスタンスの中にあるんだ。ただ、**親クラスのrun()には呼び出しが届かない**から「上書きされたように見える」だけなんだよ。

11.2.3 親インスタンス部へのアクセス

先輩、親インスタンス部のrun()は、どうせ外部から呼び出せないんですし、存在価値がないように思いますけど？

いや、親インスタンスのメソッドが役立つこともあるんだよ。

頻度として多くはありませんが、内側の親インスタンス部に属するメソッドが大活躍することもあります。たとえば、次のような例を考えてみましょう。

SuperHeroの追加仕様

SuperHeroは、空を飛んでいる状態でattack()すると、Heroでは1回だった攻撃を2回連続で繰り出すことができる。

この条件に従うために、次のようなオーバーライドを思いつくかもしれません。

リスト11-8

SuperHero.java

```
1  public class SuperHero extends Hero {
2    public void attack(Matango m) {
3      System.out.println(this.name + "の攻撃！");
4      m.hp -= 5;
5      System.out.println("5ポイントのダメージをあたえた！");
```

1回目の攻撃（3～5行目）

```
6      if (this.flying) {
7        System.out.println(this.name + "の攻撃！");
8        m.hp -= 5;
9        System.out.println("5ポイントのダメージをあたえた！");
10     }
11   }
12   ⋮
13 }
```

2回目の攻撃

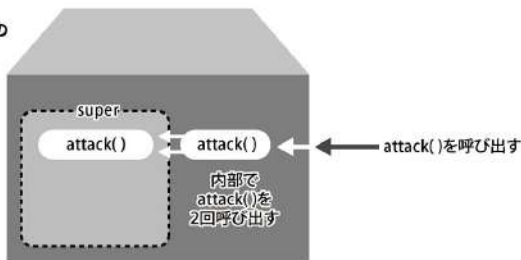
しかし、この方法では、将来 Hero クラスの attack() メソッドの処理内容が変わった場合に困った事態に陥ります。

たとえば、Hero クラスの attack() メソッドが修正され、1 回の攻撃で敵に与えるダメージが 10 に修正されたとしましょう。SuperHero インスタンスを生み出し、fly() を呼び出した後で attack() を呼び出したらどうなるでしょうか？

飛んでいるスーパーヒーローは、Hero の攻撃を 2 回するわけだから、「10 ポイントダメージの攻撃を 2 回」になるべきですよね？

残念ながら「5 ポイントダメージの攻撃が 2 回」のままなんだ。SuperHero クラスでオーバーライドしちゃってるからね。

このような場合には、次のような呼び出しを実現できれば目的を果たせます。

図 11-7　親インスタンスのメソッドを呼び出す

Javaのコードでは、次のように記述することで実現できます。

リスト 11-9

SuperHero.java

```
1  public class SuperHero extends Hero {
2    ⋮
3    public void attack(Matango m) {
4      super.attack(m);            ← 親インスタンス部の attack() を呼び出し
5      if (this.flying) {
6        super.attack(m);          ← 親インスタンス部の attack() を呼び出し
7      }
8    }
9    ⋮
10 }
```

superとは、「親インスタンス部」を表す予約語です。これを利用すれば、親インスタンス部のメソッドやフィールドに子インスタンス部からアクセスすることができます。

親インスタンス部のフィールドを利用する

super. フィールド名

親インスタンス部のメソッドを呼び出す

super. メソッド名 (引数)

superを付けずに単に「attack()」と呼び出してはダメなんですか？

ダメだ。super を付けないということは「this.attack()」と同じ意味になるね。それでは図 11-8 のように、attack() を呼び出し続ける無限ループになってしまうよ。

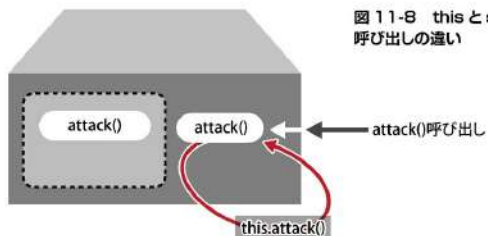



図 11-8 this と super による呼び出しの違い

「祖父母」インスタンス部へのアクセスは不可能!

A クラス(祖父母)、B クラス(親)、C クラス(自分)という 3 つのクラスが継承関係にあるとき、そこから生成されたインスタンスも 3 重構造になりますが、その際に一番外側に相当する C インスタンスについて考えましょう。

C クラスのメソッドは、一番外側のインスタンス部(自分自身)へは this、そして親インスタンス部へは super でアクセスできます。しかし、残念ながら「祖父母」にあたるインスタンス部へアクセスする手段は準備されていません。つまり、C クラスのメソッドが A クラスのインスタンス部に直接アクセスすることはできないのです。

11.3 継承とコンストラクタ

11.3.1 継承を利用したクラスの作られ方

前節では、インスタンス化されたSuperHeroがどのような姿なのかを紹介しました。その姿は、Heroインスタンスを内部に持つ多重構造になっていることを理解できたと思います。この多重構造は、クラスがnewされた際に以下のような段階を経て構築されます。

図 11-9　インスタンスの多重構造が構築される手順

図11-9の③にあるように、SuperHeroインスタンスが完成すると、JVMは自動的にSuperHero()コンストラクタを呼び出します。ここで次のようなコードを書いてみると、興味深い内部動作を確認できます。

リスト 11-10

Hero.java

```
1  public class Hero {
2    ⋮
3    public Hero() {
```

```
4      System.out.println("Heroのコンストラクタが動作");
5    }
6   ⋮
7  }
```

SuperHero.java

```
1  public class SuperHero extends Hero {
2   ⋮
3    public SuperHero() {
4      System.out.println("SuperHeroのコンストラクタが動作");
5    }
6   ⋮
7  }
```

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      SuperHero sh = new SuperHero();
4    }
5  }
```

実行結果

```
Heroのコンストラクタが動作
SuperHeroのコンストラクタが動作
```

あれ？ SuperHero インスタンスが完成したら JVM は SuperHero のコンストラクタを呼び出すはずよね？

その前に、どうして Hero のコンストラクタも動いているんだろう？

SuperHeroをnewすることでSuperHero()コンストラクタが動作するのは理解できますが、なぜか内側インスタンスのHero()コンストラクタも勝手に動作しています。

実はJavaでは、「**すべてのコンストラクタは、その先頭で必ず内部インスタンス部（＝親クラス）のコンストラクタを呼び出さなければならない**」というルールになっています。

同じクラスの別コンストラクタを呼び出すための「this()」（9.2.7項）に似た「super()」という記述で親クラスのコンストラクタを呼び出すことができます。

親クラスのコンストラクタの呼び出し

super(引数);

※ただし、コンストラクタの最初の行にしか記述できない。

よって、本来SuperHeroコンストラクタは、以下のような書き方をしなければなりません。

```
public SuperHero() {
  super();
  System.out.println("SuperHeroが生成されました");
}
```


もしプログラマがコンストラクタの一行目にsuper()を書いていない場合、コンパイラによって**「super();」という行が自動的に挿入**されます。リスト11-10のプログラムでは、この「暗黙のsuper()」が、自動的にHero()コンストラクタを呼び出していたというわけです。

図11-10　super()によるコンストラクタの呼び出し

つまり、コンストラクタは内側のインスタンス部分のものから順に呼ばれていくということですね。

そのとおり。ちなみに親インスタンス部のコンストラクタを呼び出す super() は、前節で学習した「super. メンバ名」とはまったく関係がないので混同しないよう注意してほしい。

11.3.2 親インスタンス部が作れない状況

このように、インスタンスが構築・初期化される手順を理解すると、ある条件で困ったことが発生します。次のリスト 11-11 を題材に考えてみます。

リスト 11-11

Item.java

```
1  public class Item {
2    private String name;
3    private int price;
4    public Item(String name) {
5      this.name = name;
6      this.price = 0;
7    }
8    public Item(String name, int price) {
9      this.name = name;
10     this.price = price;
11   }
12 }
```

引数 1 つのコンストラクタ（4行目）
引数 2 つのコンストラクタ（8行目）

Weapon.java

```
1  public class Weapon extends Item { … }
```

Item を継承し Weapon を定義

Main.java

```
1  public class Main {
```

```
2    public static void main(String[] args) {
3      Weapon w = new Weapon();
4    }
5  }
```

このコードではエラーが発生しますが、その理由を1ステップずつ整理しながら解説していきましょう。

Main.javaの3行目のnew Weapon()により、JVMはWeaponインスタンスを生成しようとします。WeaponクラスはItemクラスを継承していますので、このインスタンスは、内部にItemインスタンスを含む多重構造になっているはずです。

二重構造のインスタンスを作り終えると、JVMは自動的にWeapon()コンストラクタを呼び出そうとします。しかし、Weaponクラスにはコンストラクタが定義されていないため、暗黙的に次のような「デフォルトコンストラクタ(9.2.6項)」が定義され動作します。

```
public Weapon() {
}
```

しかし、ここで前項で学んだことを思い出してください。すべてのコンストラクタの先頭行には実は「super();」が隠れていますので、実際には次のようになります。

```
public Weapon() {
  super();
}
```

このように自動生成されたWeaponクラスのコンストラクタは、親クラスItemのコンストラクタを**引数なしで**呼ぼうとします。ここで、呼び出される側のItemクラスのコンストラクタの宣言を見てみましょう(リスト11-11)。引数1つのもの(Item.javaの4行目)と2つのもの(同8行目)、あわせて2つのコンストラクタが宣言されていますが、**引数が0個のコンストラクタは存在しません。**

11章

つまり、Item クラスのコンストラクタ呼び出しには、必ず引数が 1 つか 2 つ必要であり、Weapon クラスのコンストラクタからであっても「super();」のように**引数がない呼び出しはできない**のです。

11.3.3 内部インスタンスのコンストラクタ引数を指定する

このように、内部インスタンスの初期化を行うコンストラクタ（Item() コンストラクタ）に引数を与える必要がある場合は、super() の呼び出し時に明示的に引数を渡します。

Weapon.java

```
1  public class Weapon extends Item {
2    public Weapon() {
3      super("ななしの剣");
4    }
5  }
```

引数 1 つの親クラスコンストラクタを呼び出す

これで、Weapon クラスのインスタンス化によって内部で Item インスタンスが作られる際、リスト 11-11 の Item.java の 4 行目のコンストラクタが動作し、常に「ななしの剣」という名前になります。「super(" ななしの剣 ",300);」と記述すれば、常に 2 つの引数を持つ 8 行目の Item コンストラクタが動作するでしょう。

つまり、super() に与える引数の数と型によって、「親インスタンス部が初期化されるときに利用されるコンストラクタ」を明示的に指定できるのです。

図 11-11　引数によって指定した親コンストラクタの呼び出し

11.4 正しい継承、間違った継承

11.4.1 is-a の原則

継承に「正しい」とか「間違ってる」なんてあるんですか？　ボクのプログラムで誤った継承の使い方をしていないか心配になってきました…。

大丈夫。簡単なチェック方法があるよ。

正しい継承とは、**「is-a の原則」といわれるルールに則っている継承**のことです。そして is-a の原則とは、子クラスと親クラスの間に次のような概念的な関係が成立しているべきであるとする原則です。

is-a の関係

子クラス **is-a** 親クラス（子クラスは、親クラスの**一種である**）

A is-a B は日本語で「A は B の一種である」という意味であり、この文章の意味が自然であれば正しい継承です。スーパーヒーローは「特殊能力を持った特別なヒーロー」ですが、あくまでヒーローの一種であることには間違いありません。

図 11-12　SuperHero は Hero の一種である

逆に、「(子クラス) is-a (親クラス)」「(子クラス) は (親クラス) の一種である」という文章を作って不自然さを感じたら、継承の誤りを疑いましょう。

11.4.2 間違った継承の例

現実世界の登場人物同士に**概念として is-a の関係がないにもかかわらず、継承を使ってしまうのが「間違った継承」**です。例を挙げてみましょう。

ここに「名前」と「値段」のフィールドを持つ Item クラスがあります。このクラスは、勇者たちが冒険のために持ち歩く「薬草」や「ポーション」などのアイテム(小道具)を表すクラスです。そして今、私たちは新たに House クラスを作ろうとしています。House クラスには、所有者や床面積、間取りや住所などのほか、「家の名前」「家の値段」のフィールドも必要です。

Item クラスを継承して House クラスを作ればいいと思いますけど？

いや、その発想こそが「間違った継承」の原因だよ。

Item クラスを継承して House クラスを作ることは原理上、可能です。実際、名前と値段のフィールドも継承され、問題なく動作します。ですが、「House is-a Item (家はアイテムの一種である)」という文章には違和感を覚えませんか？勇者は冒険のために家を持ち歩くことはありません。

このように、「フィールドやメソッドが流用できるから」という安易な理由で継承をしてはいけません。「動くか動かないか、便利か便利でないか」ではなく、**is-a であるかどうかに基づいて**、継承は利用すべきです。

継承の利用に関するルール

is-a の原則が成立しないならば、ラクができるとしても継承を使ってはならない。

11.4.3 間違った継承をすべきでない理由

便利だからいいじゃないですか。なんでダメなんですか？

is-a の関係ではない継承を使ってはならない理由は 2 つあります。

- **将来、クラスを拡張していった場合に現実世界との矛盾が生じるから。**
- **オブジェクト指向の 3 大機能の最後の 1 つ「多態性」を利用できなくなるから。**

多態性については第 13 章で解説するとして、ここでは「現実と矛盾が生じていくこと」について、House クラスと Item クラスの例を用いて解説しましょう。

確かに House クラスを作った時点では、Item クラスを継承していても問題がないように思えます。しかし、これは単に「たまたま現時点では実害がないだけ」であって、より忠実に現実世界の家やアイテムをまねようとクラスを改良していくと次々と矛盾が生じます。

たとえば、アイテムは敵に投げつけてダメージを与えることができるとしましょう。そこで、Item クラスに「敵に投げつけたときにあたえるダメージを返すメソッド」、getDamage() を追加します。

Item.java

```
public class Item {
  :
  public int getDamage() {
    return 10;
  }
  :
}
```

11章

このメソッドは継承され House クラスでも利用可能になりますが、現実に沿って考えると**家を投げることなどできるわけがありません**し、そのダメージを算出する、getDamage() メソッドが House クラスに対して呼べること自体が極めて不自然です。

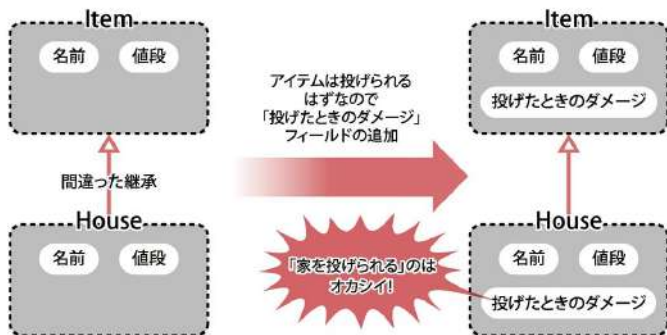



図 11-13　間違った継承を用いると、現実とは矛盾したメンバが現れる

「投げつけたときのダメージがある家」という House クラスは、すでに**現実世界の家と乖離**してしまっており、**オブジェクト指向の原則から外れています**。

なるほど…でも、「House クラスには getDamage() があるけど、無視して使わない」ことにすればいいんじゃない？

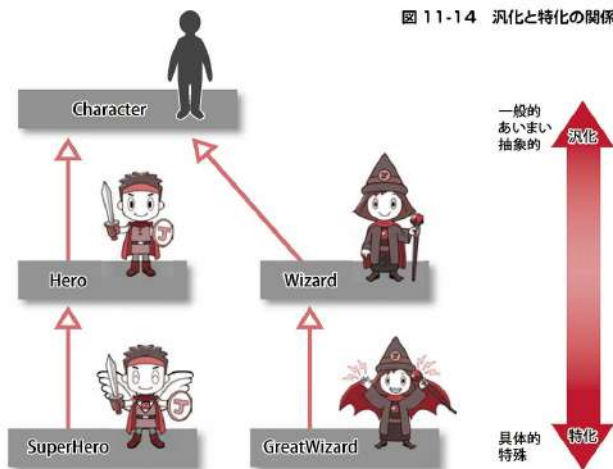
だめよ。「存在するけど実は使っちゃダメなメンバ」がいくつもあるクラスなんて、怖くて使えないわ。

11.4.4　汎化・特化の関係

正しい継承が「is-a」の関係で結ばれるということは、子クラスになるほど「特殊で具体的なもの」に具体化（**特化**）していき、親クラスになるほど「一般的で、抽象的・あいまいなもの」に一般化（**汎化**）していくことになります。

特化すればするほど、より詳細にフィールドやメソッドを定めることができ、メンバは増えていきます。逆に汎化すればするほど、フィールドやメソッドを多く定めることは難しくなってきます。

たとえば**キャラクターであれば、どんなものでも必ず名前と HP は持っている**でしょうから、Character クラスには name と hp フィールドぐらいは定義できます。

図 11-14　汎化と特化の関係

より具体的な魔法使いになると、最低でも MP を持っていて火の玉ぐらいは放てるはずであり、クラス定義には mp フィールドや fireball() メソッドが加わるでしょう。さらに具体的な「ひとにぎりの大魔法使い（GreatWizard）」は雷を落とす lightning() メソッドなど、Wizard が持っていないメソッドも持つでしょう。

ちなみに、クラス図において継承関係を表す矢印は「クラスが汎化していく方向」を表すための矢印なんだ。

なるほど。だからクラス図の矢印は、継承の方向（特化の方向）とは逆向きに描かれていたのね。

これまでは、継承のことを「コードの重複記述を減らすための道具」と捉えて学習してきたと思います。しかし継承は、**「ある 2 つのクラスに特化・汎化の関係があることを示す」ための道具**でもあるのです。

第11章のまとめ

この章では、次のようなことを学びました。

継承の基礎

- extendsを使うことで、既存のクラスに基づき新たにクラスを定義できる。
- 親クラスのメンバは自動的に子クラスに引き継がれるため、子クラスでは差分だけを記述すればよい。
- 親クラスに宣言が存在するメソッドを、子クラスで上書き宣言することをオーバーライドという。
- final付きクラスは継承できず、final付きメソッドはオーバーライドできない。
- 正しい継承とは「子クラス is-a 親クラス」の文章に不自然がない継承である。
- 継承には、「抽象的・具体的」の関係にあることを定義する役割もある。

インスタンスの姿

- インスタンスは内部に親クラスのインスタンスを持つ多重構造をとる。
- より外側のインスタンス部に属するメソッドが優先的に動作する。
- 外側のインスタンス部に属するメソッドは、superを用いて内側インスタンス部のメンバにアクセスできる。

コンストラクタの動作

- 多重構造のインスタンスが生成されると、JVMは自動的に一番外側のコンストラクタを呼ぶ。
- すべてのコンストラクタは、先頭で「親インスタンス部のコンストラクタ」を呼び出す必要がある。
- コンストラクタの先頭にsuper()がなければ、暗黙的に「super();」が追加される。

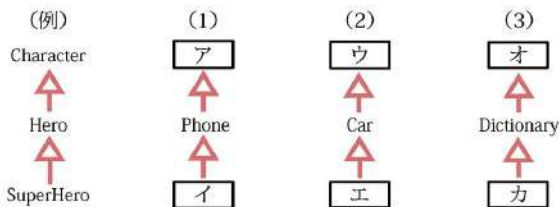
11.6 練習問題

練習 11-1

次の中から「誤った継承」であるものをすべて選んでください。

①スーパークラス:Person　サブクラス:Student
②スーパークラス:Car　サブクラス:Engine
③スーパークラス:Father　サブクラス:Child
④スーパークラス:Food　サブクラス:Susi
⑤スーパークラス:SuperMan　サブクラス:Man

練習 11-2

次のクラスに対する「親クラス」と「子クラス」を1つずつ考案して自由に挙げてください。

練習 11-3

次のようなクラス Matango があります。

Matango.java

```
1  public class Matango {
2    int hp = 50;
3    private char suffix;
4    public Matango(char suffix) {
5      this.suffix = suffix;
```

```
6     }
7     public void attack(Hero h) {
8       System.out.println("キノコ" + this.suffix + "の攻撃");
9       System.out.println("10のダメージ");
10      h.setHp(h.getHp() - 10);
11    }
12  }
```

このクラスを利用し、次の仕様に則ったPoisonMatangoクラスを作成してください。

ア．お化け毒キノコ（PoisonMatango）は、お化けキノコ（Matango）の中でも特に「毒攻撃」ができるもの。

イ．PoisonMatangoは以下のコードでインスタンス化できるクラスとする。

```
PoisonMatango pm = new PoisonMatango('A');
```

ウ．PoisonMatangoは、毒を用いた攻撃が可能な残り回数をint型フィールドとして持っており、初期値は5である。

エ．PoisonMatangoは、attack()メソッドが呼ばれると次の内容の攻撃をする。

①まず、「通常のお化けキノコ同様の攻撃」を行う。

②「毒攻撃の残り回数」が0でなければ、以下を追加で行う。

③画面に「さらに毒の胞子をばらまいた！」と表示。

④勇者のHPの1/5に相当するポイントを勇者のHPから引き、そのポイントを示すよう「～ポイントのダメージ」と表示する。

⑤「毒攻撃の残り回数」を1減らす。

11.7 練習問題の解答

問題 11-1 の解答

誤っているものは②、③、⑤です。

②・・・エンジンは車の「一部」であり、両者は has-a の関係 (p.341) にあります。

③・・・継承では親クラスや子クラスという用語を用いますが、概念としての親や子とは関係ありません。「子どもは父親の一種」ではありません。

⑤・・・スーパーマンは人間の一種ですので、スーパーという用語が付いていても、サブクラス(子クラス)です。

問題 11-2 の解答

以下は解答例です。このほかにも多数考えられます。

(ア) Device(装置)、Tool(道具)など。
(イ) MobilePhone(携帯電話)、SmartPhone(スマートフォン)など。
(ウ) Vehicle(乗り物)、Property(資産)など。
(エ) SportsCar(スポーツカー)、HybridCar(ハイブリッドカー)など。
(オ) Book(書物)、InformationSource(情報源)など。
(カ) EJDictionary(英和辞典)、Encycropedia(百科事典)など。

問題 11-3 の解答

以下は解答例です。

PoisonMatango.java

```
1  public class PoisonMatango extends Matango {
2    private int poisonCount = 5;
3    public PoisonMatango(char suffix) {
4      super(suffix);
5    }
6    public void attack(Hero h) {
```

```
 7      super.attack(h);
 8      if (this.poisonCount > 0) {
 9        System.out.println("さらに毒の胞子をばらまいた");
10        int dmg = h.getHp() / 5;
11        h.setHp(h.getHp() - dmg);
12        System.out.println(dmg + "ポイントのダメージをあたえた！");
13        this.poisonCount--;
14      }
15    }
16  }
```

第 12 章

高度な継承

私たちは現実世界において、
無意識に多くのものを抽象的に捉え、利用しています。
オブジェクト指向の目的が「現実世界の再現」である以上、
Java でも「抽象的な登場人物」を上手に扱える必要があります。
この章では、Java 仮想世界においても、抽象的であいまいなクラスを
正しく・安全に・便利に利用するために準備されている
クラスの定義方法を紹介します。

CONTENTS

未来に備えるための継承

12.1.1 高度な継承を学ぶにあたって

第 11 章ではオブジェクト指向の花形「継承」について学びました。そのメリットを十分に実感できたのではないでしょうか。また、章の最後では、正しい継承が「抽象的なクラスと具体的なクラスの関係を結ぶ」ことも学びました。図 11-14 (p.437) のような継承ツリーを親クラス、その親クラス、さらにその親クラス…と辿っていくほど、クラスはあいまいで抽象的なものになるのでしたね。

この第 12 章では、主にこのツリーの上側に登場する**「あいまいなクラスたち」の定義方法**について学びます。

これまでに学んできた通常の方法でこれらのクラスを定義しても、プログラムは動作します。しかし、「あいまいなクラスたち」専用のクラス定義方法をマスターし、高度な継承を実現することで、より安全で便利にクラスを利用できるようになるのです。

具体的には「抽象クラス」や「インタフェース」というものを学んでいくよ。

なんだか名前からして難しそうですね…。

確かに難しそうなイメージがあるけど、コツを 1 つ知っておくだけでグンと楽になるから安心してほしい。

残念なことに、「普通の継承はすぐ理解できたのに、高度な継承でつまずく」という人も珍しくありません。実は「高度な継承を学習するには、**ある意識を今までと切り替える必要がある**」というコツがあるのです。

そこで章の始めに、まずこの「意識の切り替え」について紹介していきます。

抽象クラスやインタフェースの理解に自信がないという人、そして過去の学習で挫折したという人も、このコツをおさえた上でぜひ再チャレンジしてほしい。きっとマスターできるはずだ。

12.1.2 新しい「立場」で考える

この章で学ぶ高度な継承は、文法的に難しいものではありません。にもかかわらず、なぜつまずく人が多いかというと、**高度な継承を使うときの「立場」が、今までの「立場」とはまったく違う**ということを意識せず学習を始めてしまうからなのです。

今までみなさんは、自分がどのような「立場」でJavaを使うことをイメージしてきましたか？ その多くは、作る必要がある（または作りたい）プログラムは明確に決まっていて、**そのプログラムのためだけに必要なクラスを作って目的のプログラムを完成させる「立場」**ではないでしょうか。

そして、もし開発すべきクラスと類似した既存のクラスがあれば、**継承を利用して子クラスを作る**ことにより、ゼロから開発することなく、いくつかのメンバを追加するだけで効率よくクラスを開発できるのでしたね。

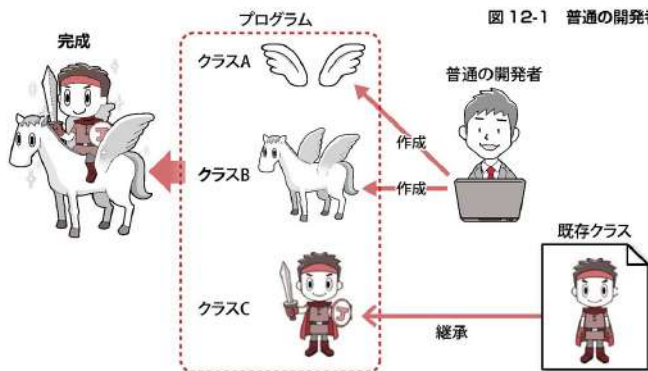



図 12-1 普通の開発者の立場

ここで「既存クラス」に注目して、少し想像を膨らませてみましょう。きっとこの**クラスを事前に開発しておいてくれた開発者**がどこかにいるはずです。

その作者は、自分の作ったクラスがどんなプログラムに利用されるか想像もつかない過去の段階で、「いつか誰かが、このクラスを継承して開発したら便利だろう」と未来に思いを馳せ、**継承の材料**となる既存クラスを作ってくれたのです。とてもありがたいことですね。

ここで次の図 12-2 を見ると、異なる「立場」で活躍する 2 種類の開発者がいることがわかります。

立場 1：現在、目の前のプログラム開発に必要なクラスを作る開発者（既存クラスを継承し子クラスを作る）。
立場 2：未来に備え、別の開発者が将来利用するであろうクラスを準備しておく開発者（親クラスとなるクラスを作っておく）。

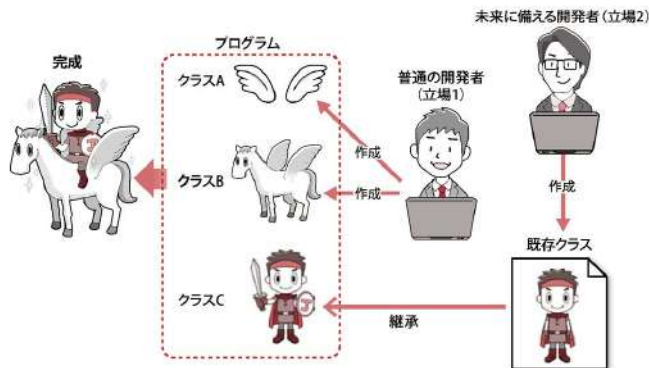



図 12-2　未来に備え既存クラスを作る開発者の立場

この章の内容をスムーズに理解するためには、この **2 つの立場**の存在を明確に意識・区別できるかがポイントです。

前章までに私たちが学習してきた知識はすべて、**立場 1** としてプログラム開発をするために必要なものでした。一方、この章で学習する知識は、みなさんが**立場 2** としてプログラムを作るときに必要となるものです。なぜなら、この章で学ぶ「抽象クラス」「インタフェース」とは、立場 2 の人たちが、**「立場 1 の人たちに安全・便利に使ってもらえる親クラスを作る」ための道具**だからです。ぜひ、立場 2の開発者になってクラスを作ることを想像しながら本章を読み進めてください。

なるほど。でも、ボクのような新入社員が入社直後に、いきなり「未来に備える大事な立場」を任されることはなさそうですね。

抽象クラスやインタフェースを自ら作ることはなくても、継承元として利用することはあるから、マスターしておく必要はあるよ。

12.1.3 「未来に備える開発者」の立場の具体例

たとえばゲーム開発の例を考えてみましょう。プロジェクトではAさん、Bさん、Cさん、そしてあなたの4人で開発を進めていますが、ある日「開発効率が悪い」ということが問題になりました。

そこで「プロジェクト全体の開発効率を改善する責任者」に任命されたあなたが調査したところ、「開発者がそれぞれ、HeroやWizardなどの似たクラスをイチから作っている」ということに気づきます(図12-3)。

開発者A　開発者B　開発者C

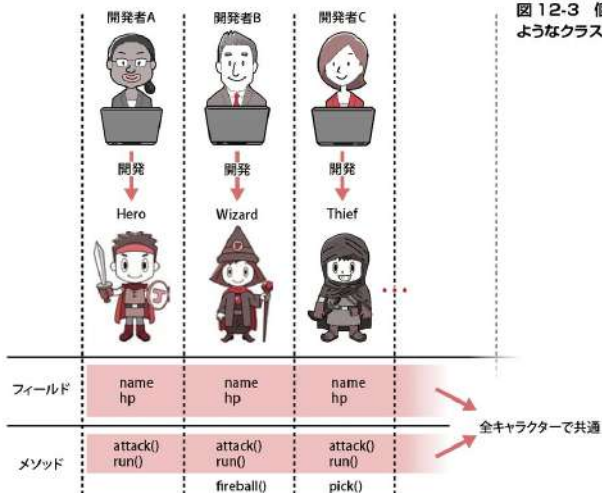
開発　開発　開発

Hero　Wizard　Thief

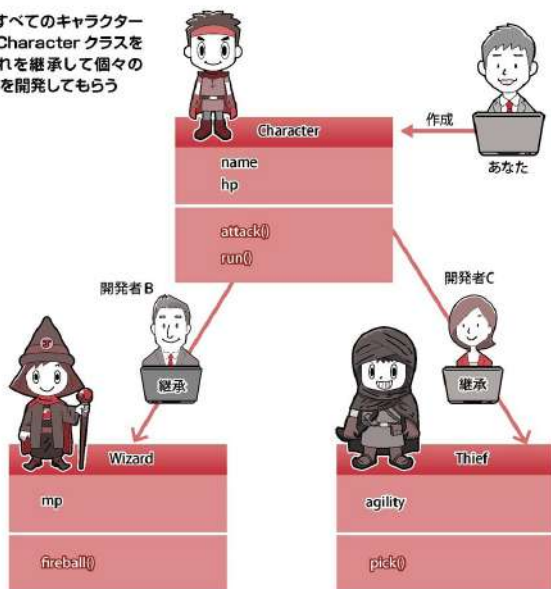
	Hero	Wizard	Thief
フィールド	name hp	name hp	name hp
メソッド	attack() run()	attack() run()	attack() run()
		fireball()	pick()

全キャラクターで共通

図12-3　個々の開発者が同じようなクラスを作っていた

12章

各キャラクターのクラスでは、name や hp などのフィールドと、attack()、run() などのメソッドは共通ですので、それぞれ別に開発するのはムダです。また、今後のバージョンアップにより「商人」や「占い師」など、さまざまなキャラクターが増える予定ですが、それらも一から開発していては効率が悪そうです。

そこで、あなたは各クラスに共通するフィールドやメソッドを持つ Character クラスを準備し、各開発者に対しては「みなさんは私の作った Character クラスを継承して、独自のフィールドやメソッドを付け足すだけで大丈夫ですよ」とアナウンスします（図 12-4）。

図 12-4　すべてのキャラクターの元となる Character クラスを準備し、それを継承して個々のキャラクターを開発してもらう

これなら各開発者は、Hero や Wizard など、それぞれの職業に特有なフィールドやメソッドだけを開発すれば済むので効率的ですね。

このときのあなたは、A さん、B さん、C さんのような「今すぐ必要な、実際に利用されるクラス」を作っている開発者（立場 1）ではなく、「未来に備えて、継承元となるクラス（継承の材料）」を作るという立場（立場 2）にあります。

開発プロジェクトの最前線で今すぐ必要な Hero、Wizard クラスの開発をがんばる A さん、B さん、C さんに対して、あなたは後方から開発支援の道具（＝共通部分まで事前に作っておいた Character クラス）を供給して援護していると捉えてもいいでしょう。

あなたの作る、たった 1 つのクラスの優劣が、それを利用する複数の開発者たちの開発効率に影響するわけです。立場 2 のあなたが意識すべきことは、「**立場 1 の開発者が効率よく安心して利用できる継承の材料をいかに作るか**」ということなのです。

「未来に備える開発者」の役割

ほかの開発者が効率よく安心して利用できる継承の材料を作ること。

未来のために継承の材料を作っておくという立場では、ほかの開発者や未来の開発者に少しでもラクをしてもらいたい、という思いやりが大事なんですね。

12.2 高度な継承に関する2つの不都合

12.2.1 2つの不都合、3つの心配

効率アップのためにCharacterクラスを作れば一件落着ですね。

いやいや、ところが落着しないんだ。湊くんが本当に、Aさんや B さんがラクになれるよう心を砕いて Character クラスを作ろうとすると、いくつか「不安」が出てくるはずなんだ。

えっ！ どうしてですか？

立場 2 として何を意識すべきかを真剣に考えないなら、単に Character クラスを作って終わりにすることもできます。

しかし、「他の開発者がラクできるように心を砕くこと」を強く意識して Character のようなクラスを開発していると 2 つの「不都合」に直面します。その不都合を原因とする、さまざまな「心配」も出てくることでしょう。「抽象クラス」や「インタフェース」は、この不都合や不安を解決してくれる道具です。

それでは、図 12-5 のような関係にある、それぞれの不都合と不安を 1 つずつ見ていきましょう。

図 12-5　継承の材料となるクラスに関する「2つの不都合」と「3つの心配」

12.2.2 最初の不都合

まずは最初の不都合Aを体験するため、実際にCharacterクラスを作成してみましょう。

リスト12-1

Character.java

```
public class Character {
  String name;
  int hp;
  // 逃げる
  public void run() {
    System.out.println(this.name + "は逃げ出した");
  }
  // 戦う
  public void attack(Matango m) {
    System.out.println(this.name + "の攻撃！");
    m.hp -= ??;
    System.out.println("敵に？？ポイントのダメージをあたえた！");
  }
}
```

ここを記述しようとして手が止まる

Characterクラスを実際に書こうとすると、attack()メソッドの内容に差しかかったところで手が止まってしまうはずです。

そうなんです…。「attack()メソッドを書かなきゃいけない」んだけど、「ダメージを何ポイントと書いたらいいのかわからない」というか…。

なぜ「手が止まってしまったか」を、より深く考えてみましょう。このCharacterクラスは将来、さまざまな開発者によって継承され、HeroやWizardやDancerなどを開発する際の材料として利用されます。

しかし、未来に完成するであろうHeroやWizard、そしてDancerは、それぞれお化けキノコを攻撃したときに与えるダメージが違うはずです。腕っ節の強いHeroであれば与えるダメージは10ポイント、ひ弱なWizardなら5ポイント、さらに今は存在しませんが、未来に追加されるかもしれない強力なキャラクターでは100ポイントなどもありえます。

つまり、Characterクラスを作っている時点では、まだ**「attack()メソッドの内容を確定できない」**ため、**書きようがない**のです。

不都合A

継承の材料となるクラスを作る時点では、その処理内容をまだ確定できない**「詳細未定メソッド」**が存在する。

12.2.3　不都合Aに対する間違った解決方法

それではattack()メソッドのような内容を確定できないメソッドを、どのように記述すればよいのでしょうか？ まず考えつくのが、Characterクラスに、そもそもattack()メソッドを記述しないようにするという方法です（図12-6）。HeroやWizardなどの新しいキャラクタークラスを作成する際には、それぞれ継承先のクラスでattack()メソッドを追加してもらいます。

しかし、この方法では他の開発者が将来新しいキャラクタークラスを作成する際、継承先のクラスにattack()メソッドを追加し忘れると「攻撃できないキャラクター」ができてしまいます。

そもそも「現実世界（ゲームの世界）のすべてのキャラクターはHP属性を持ち、攻撃ができる」という前提でCharacterクラスを作り始めたのに、「attack()を持たないキャラクター」ができてしまっては困ります。

うーん。でも、そんなことは「必ずattack()を作るように各開発者自身が気をつければいい」んじゃないですか？

Character
name
hp
run()
作成
自分
自分は attack()は 作らない
継承
Wizard
mp
attack()
fireball()
それぞれの 職業クラスで attack()を作る
未来の開発者

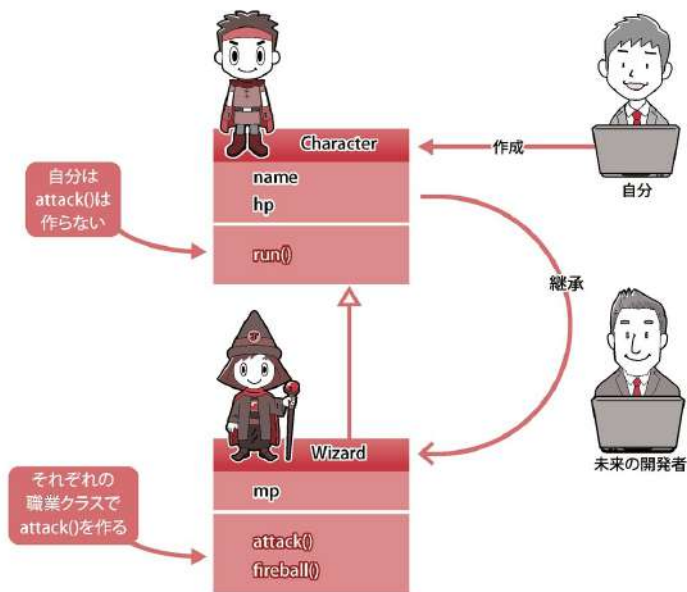



図 12-6　それぞれ継承先のクラスで attack() メソッドを追加してもらう

第 10 章でも説明したとおり、人間は必ずミスをする。その責任を人に求めるのではなく「ミスを防ぐしくみ」を考えるべきなんだ。

そもそもオブジェクト指向とは、「現実世界（今回の場合はゲームキャラクターたちが住む世界）を正確に写し取る」ことでした。そして、**現実世界と Java コードの世界に矛盾が生じる余地があるから不具合が生じる**のです。

「キャラクターであれば少なくとも攻撃ができるはず」という前提を考えると、「攻撃できないキャラクターが作れてしまうこと」は万が一にもあってはなりません。**Character クラスは必ず attack() メソッドを持っているべき**なのです。

現実世界に対応したメソッド定義の必要性

「現実世界の登場人物が持つ操作」なのであれば、クラスのメソッドは存在しているべきである（仮に、メソッドの処理内容は確定困難であったとしても）。

12.2.4 不都合Aに対する対応策と2つの心配

不都合Aの対応策として、「Characterクラスのattack()メソッドは内容を確定できないので、とりあえず空にしておこう」と思いつくかもしれません。他の開発者がattack()メソッドを継承して、それぞれの職業クラスを作成する際に、その職業に最適なattack()メソッドでオーバーライドしてもらう、という考え方です。

リスト12-2　attack()メソッドの中身を空にしておく

Character.java

```
1 public class Character {
2   String name;
3   int hp;
4   public void run() {
5     System.out.println(this.name + "は逃げ出した");
6   }
7   public void attack(Matango m) {
8   }
9 }
```

メソッドの中身を空にしておく

リスト12-3　未来の開発者が開発するコード

Characterのattackメソッドをオーバーライドする

Hero.java

```
1 public class Hero extends Character {
2   public void attack(Matango m) {
```

```
3      System.out.println(this.name + "の攻撃！");
4      System.out.println("敵に10ポイントのダメージをあたえた！");
5      m.hp -= 10;
6    }
7  }
```

なかなか悪くない方法です。しかし、あなたの「Characterクラスを利用してくれる未来の開発者たち」のことを思いやる気持ちが強いほど、次の2つの心配が頭をよぎるでしょう。

図 12-7　詳細未定メソッドに関する2つの心配

12.2.5 第1の心配事：オーバーライドし忘れ

未来の開発者がHeroやWizardなど具体的な職業のクラスを作る際にattack()のオーバーライドを忘れてしまうと、重大な不具合に直結します。たとえばHeroクラスを作る際、attack()メソッドをオーバーライドし忘れたとします。

リスト 12-4

Hero.java

```
1  public class Hero extends Character {
2  }
```

attack() をオーバーライドすべきなのにしていない

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h = new Hero();
4      Matango m = new Matango();
5      h.attack(m);
6    }
7  }
```

メソッドは呼び出せるが…

12章

Hero クラスは親クラスである Character から**内容が空の attack() メソッド**を受け継いでいます。そのため、main() メソッドなどから「attack メソッドを**呼び出せるが、何も起きない**」という不具合を抱えたクラスになってしまいます。

main() メソッドでは「当然、何か攻撃してくれるんだろう」と思って attack() を呼び出すのに、何もしてくれないなんて…。

エラーは出ないけど「想定外の変な動きをする」というタチが悪い不具合だ。コンパイル時や実行時にエラーが起きてくれたほうが、誤りに気づけるからまだマシなんだけどね。

ボクが作った Character クラスを利用する開発者の人たちが、こんな不具合に苦しんでほしくないなぁ…。

　解決策の 1 つとして、Character クラスの attack() メソッドを作成する際、次のようにコメントを残しておくという方法があります。

```
1  // 未来の開発者さまへ
2  // 私はCharacterクラス開発者のミナトです。
3  // このクラスを開発している時点では、将来このクラスを継承して
4  // 作られるそれぞれの職業クラスが何ポイントのダメージを与えるか
5  // を確定できないため、以下のメソッドは中身を空にしてあります。
6  // Characterクラスを継承して様々な職業クラスを作る際には、
7  // attack()の中身を必ずオーバーライドして使ってください。
8  public void attack(Slime s) {
9  }
```

　しかし、Character クラスを継承する未来の開発者が、このコメントを見逃したり無視したりする可能性は残ります。仮に無視しなかったとしても、次のようなミスを犯してしまう可能性はあります。

リスト 12-5

Hero.java

```
1  public class Hero extends Character {
2    public void atack(Matango m) {
3      System.out.println(this.name + "の攻撃!");
4      System.out.println("敵に10ポイントのダメージをあたえた！");
5    }
6  }
```

attackのtが1文字足りずオーバーライドになっていない！

そっか。このクラスは継承してきた空のattack()と自分自身で定義したatack()の2つのメソッドを持ってしまうんですね。

そうだ。Heroクラスの作者は、「自分がちゃんとattack()をオーバーライドして空の処理を上書きした」と思い込んでしまっているだろう。

何も知らずに、main()メソッドとかからHeroのattack()を呼び出しちゃったら…と思うとゾッとしちゃいます。

12.2.6 第2の心配事:「本当に何もしない」と区別がつかない

Characterクラスのattack()メソッドを、もう一度よく見てください。次のようになっているはずです。

```
public void attack(Matango m) {
}
```

そもそも、この書き方は「呼ばれても何もしない」メソッドを作りたい場合に行うものです。しかし今回の場合、attack()は**「何もしない」のではなく、「何をするかが未定で記述できない」**のです。

未来の開発者がこのメソッドを見たときに、「何もしないのが正しい」のか、それとも「何をするか未定」なのか、**区別がつかないおそれがある**のです。

そういえば、「詳細未定」欄だらけの菅原さんの年間営業計画ですが…もし空白のままだったら「未定」か「何もやらない（決定）」かが区別つきませんね…。ボクも見習わせていただきます（笑）。

そんなところはマネしなくていいから！

12.2.7 第３の心配事：意図せず new して利用されてしまう

ここまで Character クラスに関する 2 つの心配事について考えました。それらはいずれも「詳細未定メソッド」に関係した心配事でしたが、まったく別の観点からの心配事がもう 1 つあります。それは、**「未来の開発者が間違って Character クラスを new して利用してしまうかもしれない」という心配**です。

たとえば、プロジェクトに新しく入った Java 初心者の D さんに、あなたが「便利なクラスだからどうぞ使ってください」と言って Character クラスを渡したら、D さんは次のようなコードを書いてしまうかもしれません。

リスト 12-6

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Character c = new Character();
4      Matango m = new Matango('A');
5      c.attack(m);
6    }
7  }
```

Hero や Wizard ではなく Character を new してしまった！

オーバーライドされていないので何も動かない！

いやいやいや！ そもそも使い方が完全に間違っていますよ！ この Character クラスは継承の材料として使われるべきものであって、Hero や Wizard みたいに new して使うためのものじゃないんです！！

そのとおり。ただ、Character クラスが作られた経緯をよく知らない人や Java 初心者は、間違って new しようとするかもしれないね。

実は、この Character クラスから実体であるインスタンスが生み出されたり、そのインスタンスが仮想世界の中で活動してしまったりすることは**かなりの異常事態**です。なぜなら、詳細未定な attack() メソッドを含む Character クラスは「詳細未定につき、作りこんでない部分が残っている未完成な設計図」のようなものだからです。

Character クラスの例に限らず、「未来の開発者のために準備しておくクラス」は、多かれ少なかれ未完成な部分が残っているものです。そのような設計図に基づいて、実体である製品（たとえば車）を生産・利用したら大変な事故につながることは容易に想像できるでしょう。

そもそも「**一部でも未完成部分が残っている設計図から、実体を生み出してはならない**」のです。

そもそも new されるべきではないクラス

Character のように、「詳細未定」な部分が残っているクラスはインスタンス化されてはならない。

12.2.8 第3の心配事の原因

それではなぜ、第3の心配事（間違ってnewされてしまう心配）が出てきてしまったのでしょうか？　今まであまり意識することはありませんでしたが、そもそもクラスには2つの「利用の仕方」があります。

① newによる利用　　：インスタンスを生み出すために、そのクラスを利用する。
② extendsによる利用：別のクラスを開発する際、ゼロから作ると効率が悪いので、あるクラスを継承元として利用する。

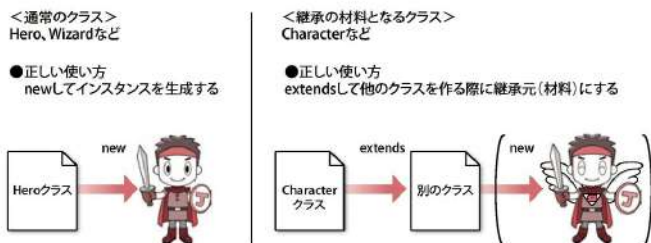



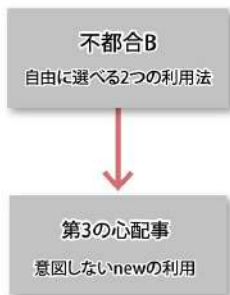
図12-8　クラスの2つの利用方法

そしてHeroやWizardはnewするためのクラスとして、またCharacterはextendsするためのクラスとしてそれぞれ開発されています（図12-8）。

図12-9　第3の心配

しかし残念ながら、Characterクラスの作者が「extendsして利用してほしい」と願っても、未来の開発者は「newによる利用」と「extendsによる利用」のどちらも選べてしまいます。ですからCharacterのような未完成なクラスが誤ってnewされてしまうという事態が起きるのです。

「クラスには自由に選べる2つの利用法がある」という利点が、皮肉にも「意図せずnewされる」という心配の原因になっています。

このような過ちが起こることがないように、ソースファイルの先頭にコメント

を書くというアイデアも考えられます。

Character.java

```
1  /*
2   * 未来の開発者さまへ
3   * 私はCharacterクラス開発者、ミナトです。
4   *
5   * このクラスは、普通のクラスのようにnewして使うためのものでは
6   * ありません。 HeroやWizardなどの職業クラスを皆様が作る際に
7   * 少しでもラクができるように、 全職業クラスに共通するフィールド
8   * やメソッドをそなえた「継承の材料」です。
9   *
10  * このクラスを継承して、必要なフィールドやメソッドを追加する
11  * ことで、それぞれの職業クラスを完成させてください。
12  * 逆に言えば、このCharacterクラスは、それ自体では未完成の
13  * クラスです。たとえばattack()メソッドは中身が確定できないので
14  * 空にしてあります。
15  * よって、このクラスをnewして実際に利用（冒険させたり
16  * 戦闘させたり）しないでください。不具合の原因になります。
17  */
18 public class Character {
19 }
```


それでも、このようなコメントも読み落とされたり無視されたりする可能性があります。どうすればよいのでしょうか？

ボクの作ったCharacterクラスが、未来の開発者によって正しくない使われ方をされて不具合の原因にならないか不安ですよ…。

しかし、湊くんの心配は無用です。なぜならJavaには、これまでに見てきた3つの心配事を解決するしくみがあるからです。次の節で、それを学んでいきましょう。

12.3 抽象クラス

12.3.1 安全な「継承の材料」を実現するために

前節では、継承の材料となるクラス（Character クラスなど）を作ろうとすると問題になる２つの不都合と３つの心配事について考えてみました。

図 12-10　２つの不都合と３つの心配事

そして Java には、この３つの心配事を解決するしくみが準備されています。この節では、まず第２の心配事と第３の心配事、そして第１の心配事の順に解決の方法を見ていきましょう。

12.3.2 詳細未定メソッド専用の書き方

まずは第２の心配事から解決していきましょう。

第２の心配事

空のメソッドを作っておくと、「現時点で処理内容を確定できないメソッド」なのか「何もしないメソッド」なのか、区別がつかない。

実は Java には、「詳細未定メソッド」を記述する専用の構文が準備されています。

詳細未定メソッド(抽象メソッド)の宣言

アクセス修飾子 abstract 戻り値 メソッド名(引数リスト);

たとえば Character クラスの attack() メソッドは次のように書きます。

リスト 12-7

Character.java

```
1 public class Character {
2   ⋮
3   public abstract void attack(Matango m);
4   ⋮
5 }
```

abstract(アブストラクト)とは「抽象的・あいまい」という意味の英単語です。これをメソッドの宣言に付けることで、「attack() というメソッドは少なくとも宣言すべきなのですが、具体的にどう動くか、内容がどうなるかまでは現時点では確定できないので、メソッド内部の処理はここでは記載しない」という表明になります。

メソッドの処理内容は未定で記載できないわけですから、メソッド宣言の後ろにはブロック記号の{}さえ付けず、その代わりにセミコロンを書きます。abstract 修飾子が付けられたメソッドは、**抽象メソッド**(abstract method)と呼ばれます。

第 2 の心配事は解決!

空メソッドは「何もしないメソッド」、抽象メソッドは「何をするか現時点で確定できないメソッド」と区別できる。

12.3.3 未完成のため new してはいけないクラスの宣言

次に第 3 の心配事を解決しましょう。

第 3 の心配事

未完成部分を含む継承専用のクラスを誤って new される可能性がある。

Java では、「未完成部分（＝抽象メソッド）を 1 つでも含むクラス」は、次の構文に従って宣言しなければならないことになっています。

抽象メソッドを含むクラスの宣言

```
アクセス修飾子 abstract class クラス名 {
  ⋮
}
```

たとえば、抽象メソッド attack() を持つ Character クラスを宣言するには、次のリストのように書きます。

リスト 12-8

Character.java

```java
public abstract class Character {
  String name;
  int hp;
  public void run() {
    System.out.println(this.name + "は逃げ出した。");
  }
  public abstract void attack(Matango m);
}
```

抽象クラスとして Character を宣言

Java のルールで、抽象メソッドを含むクラスは**必ず abstract 付きのクラスにしなければならない**。もし 1 行目の abstract を忘れてしまったらコンパイルエラーになるよ。

このように、abstract が付いたクラスは特に**抽象クラス**と呼ばれます。Character クラスを普通のクラスではなく抽象クラスとして宣言すると、次のような特殊な制約がかかります。

抽象クラスの制約

抽象クラスは、new によるインスタンス化が禁止される。

たとえば、抽象クラスとして宣言された Character をインスタンス化しようとする次のコードはコンパイルに失敗します。

```
Character c = new Character();
```

```
エラー： Character は abstract です。インスタンスを生成することはできません。
```


Character のように継承の材料となるクラスを開発する際には、抽象クラスとして宣言しておけばよいのです。

第 3 の心配事も解決！

継承専用のクラスは抽象クラスとして宣言すれば、間違って new されることはない。

抽象メソッド（＝未完成部分）が１つでもあるクラスは、抽象クラスにしなければコンパイルが通らないことも併せて考えてみよう。

「詳細未定」の抽象メソッドがある→そのクラスは必ず抽象クラスになる→インスタンス化できない…。

「一部でも未定な部分がある設計図」から実体が生まれてしまうようなこと（12.2.7 項）が絶対になくなるんですね！

12.3.4 オーバーライドの強制

最後は第 1 の心配事の解決です。

第 1 の心配事

未来の開発者が、詳細未定メソッドをオーバーライドし忘れる可能性がある。

この心配事は、実はすでに解決しています。「Character を継承して Dancer を作る新人開発者 D さんの立場」になって考えてみましょう。

Dancer は Hero や Wizard と同じように、new して実体を生み出し冒険させるためのクラスです。

リスト 12-9

Dancer.java

```
public class Dancer extends Character {

  public void dance() {
    System.out.println(this.name + "は情熱的に踊った");
  }
}
```

Character は抽象クラス

attack() をオーバーライドし忘れている

Dさんは、Dancer特有の能力である「踊る (dance)」というメソッドの作成に気を取られて attack() のオーバーライドを忘れてしまいました。

しかし、このソースをコンパイルしようとすると「**未完成状態のクラスであるDancerは、abstractを付けて抽象クラスにしなければならない**」という意味のエラーメッセージが表示され、コンパイルは失敗します。

Dancerクラスの定義には1つも抽象メソッドがないから、抽象クラスにしなくてもいいハズなのに…。

いや、Dancerには**隠れた「抽象メソッド」**が潜んでいるんだよ。

ここで継承の基本を再確認しましょう。Dancerクラスは Character クラスを親クラスとしますので、**Character クラスが持つすべてのメンバを継承**しています。そして、親クラスから継承したメンバの中には、抽象メソッド attack() も含まれています。つまり、**Dancer クラス自体のソースコードに抽象メソッドはなくても、親クラスから抽象メソッドを継承して持っている**のです。抽象メソッドが1つでもある以上、Dancer クラスも抽象クラスにしなければコンパイルエラーが出て当然ですね。

12章

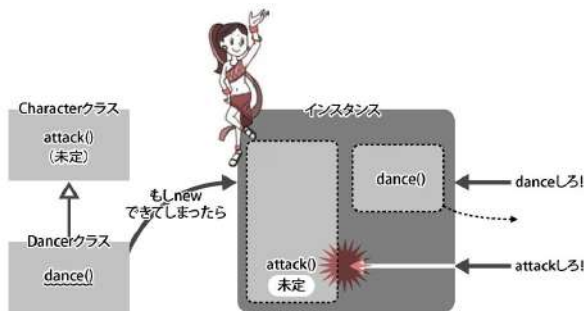



図 12-11　Dancer クラス内に抽象メソッド attack() が含まれている

このエラーに対処する方法は2つあります。

① Dancer クラスの宣言に abstract を付けて抽象クラスにする。
② Dancer クラス内部の「未完成部分」をすべてなくす。

①の方法で解決を図ればコンパイルエラーを消すことはできます。しかし、抽象クラスとなってしまった Dancer は new できませんので、Hero や Wizard のようにインスタンスを生み出すことができません。

D さんが「Dancer を Hero や Wizard のようにインスタンス化して冒険に出せるクラスとして開発したい」ならば、残された選択肢は②だけになります。すなわち Dancer クラスの中で attack() をオーバーライドし、未完成メソッドをなくしてあげればよいのです（リスト 12-10）。

リスト 12-10

Dancer.java

```
1  public class Dancer extends Character {
2    public void dance() {
3      System.out.println(this.name + "は情熱的に踊った");
4    }
5    public void attack(Matango m) {
6      System.out.println(this.name + "の攻撃");
7      System.out.println("敵に3ポイントのダメージ");
8      m.hp -= 3;
9    }
10 }
```

親から継承した「詳細未定の attack()」を上書きする

このオーバーライドによって、宣言しかされていなかった attack() メソッドの動作が決定されました。このように、それまで未定だったメソッドの内容を確定させることを、**実装**（implements）すると表現することもあります。

リスト 12-10 の Dancer クラスは、すべてのメソッドの動作が実装されており「詳細未定」な部分は残っていません。よって abstract を付ける必要はありません。Dancer は **new して使える通常のクラス**になりました。

今回の Dancer の例を振り返ってみると、「抽象クラスは new できない」というルールが Java に備わっている以上、「あるメソッドを抽象メソッドとして宣言し

ておくことで、未来の開発者にオーバーライドを強制できる(オーバーライドしないと new して使えないから)」という効果があるとも言えます。

第 1 の心配事も解決!

詳細未定なメソッドを抽象メソッドとして宣言すれば、未来の開発者にオーバーライドを強制できる。

　以上で 3 つの心配事がすべて解決しました。抽象クラスと抽象メソッドを用いることで、未来の開発者が安全かつ便利に利用できる「継承の材料」となるクラスを開発できるのです。

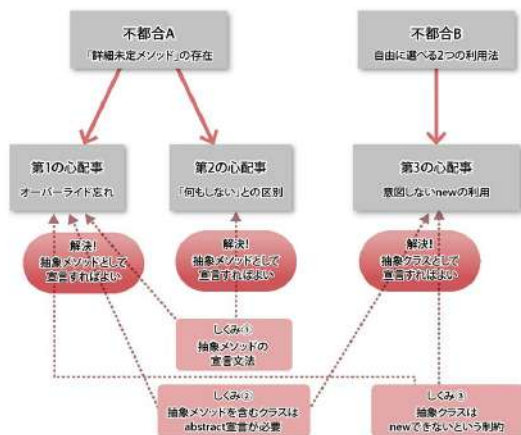



図 12-12　3 つの心配事の解決

これで安心してボクのCharacterクラスを使ってもらえますね。

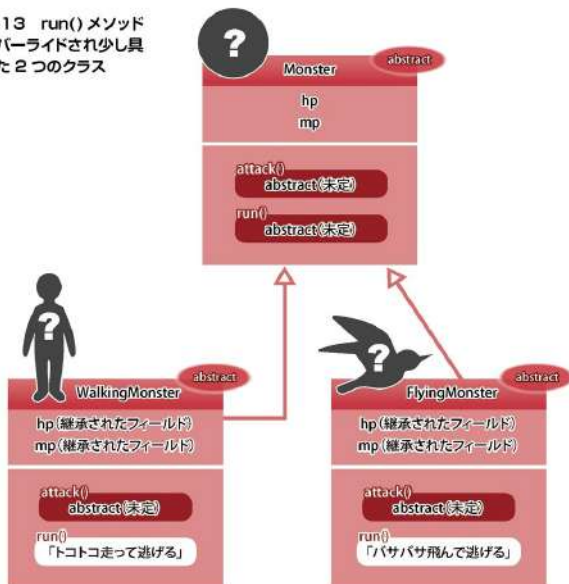
12.3.5 多階層の抽象継承構造

ここまでの解説で、「抽象メソッドには、未来の開発者が継承する際にオーバーライドを強制する効果があること」が理解できたと思います。

ちなみに、抽象クラスを継承した子クラスで「すべての抽象メソッドをオーバーライドしてメソッドの内容を実装する」必要があるわけではありません。

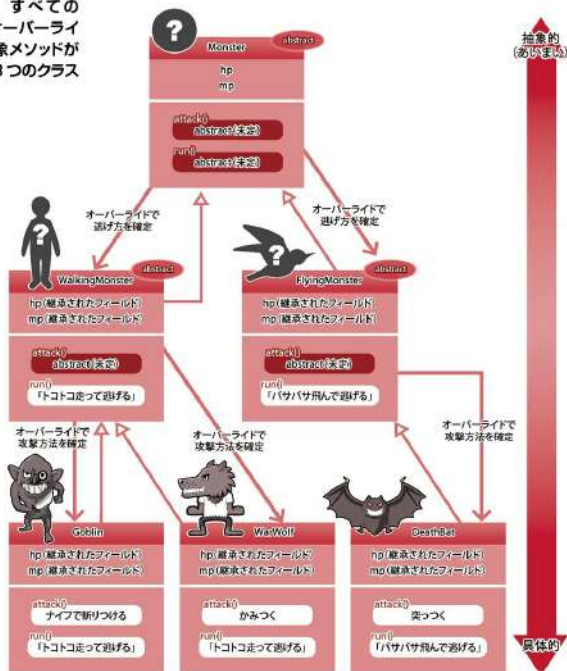
そのクラスでは確定できない抽象メソッドについては、必要に応じて、その孫クラス、あるいは曾孫クラスでオーバーライドして内容を確定させてもよいのです。その代わり、**すべての抽象メソッドの処理内容が確定しなければ abstract を外すことは許されず**、つまり **new して利用することはできません**。

たとえば、さまざまなモンスターたちの親クラスとして Monster クラスというものを考えましょう。Monster クラスは attack() と run() のメソッドを持っていますが、モンスターによって、どう攻撃するか、どう逃げるかについては、現時点ではわかりません。よって、両方とも abstract が付いた抽象メソッドです。

図 12-13　run() メソッドがオーバーライドされ少し具体化した 2 つのクラス

次に、もう少し具体的なMonsterを定義するとします。「WalkingMonster」は「トコトコ走って」、「FlyingMonster」は「バサバサ飛んで」逃げていくと決まれば、run だけはオーバーライドして内容を実装できます（図 12-13）。

しかし、WalkingMonster と FlyingMonster にはまだ抽象メソッドが残っています。attack() の詳細が未定なので、これら 2 つのクラスは共に抽象クラスとしなければならず、クラス宣言から abstract 宣言は外せません（つまり new できない）。

もし、WalkingMonster の子として Goblin クラスを定義し、このクラスでは attack() をオーバーライドすると、ここでやっと抽象メソッドがなくなります。Goblin クラスは abstract 宣言を付ける必要はなくなり、通常のクラスとして new して利用することが可能になります。同様に FlyingMonster の子として

図 12-14　すべてのメソッドがオーバーライドされ、抽象メソッドがなくなった 3 つのクラス

DeathBatを定義し、このクラスでattack()をオーバーライドすると抽象メソッドはなくなり、これもnewして利用できるようになります（図12-14）。

この継承図を眺めると、継承が繰り返されるたびに具体化していくことがわかります。Monsterというクラスは大変にあいまいで「HPとMPがある」程度しか決められていません。

もう少し具体化したWalkingMonsterやFlyingMonsterでは、逃げる処理の内容が確定します。さらに具体化したGoblinやDeathBatは、攻撃の方法も確定し、あいまいさがまったくありません。

第11章の最後でも述べたように、継承を正しく用いたJavaのクラスは、継承階層を降りていくほどに具体的になり、メソッドの処理内容が実装されていきます。

継承関係によるアクセス制御

第10章「カプセル化」において、4つのアクセス修飾子を紹介しましたが、その中で簡単にしか触れていなかったものが**protectedアクセス修飾子**（protected access identifier）です。

protectedが付いたメンバは、「**自分のクラスの子孫、または、同じパッケージからのアクセスだけが許可される**」という特徴があります。使いどころが明白で使用頻度も高いprivateやpublicと比較すると、protectedを利用する局面は少ないでしょう。

12.4 インタフェース

12.4.1 抽象階層を上に辿ると…

前節の最後では、「継承階層を下に辿っていくとどうなるか」を見ていきました。継承階層が下がっていくたびにクラスは具体化してゆき、最終的にはメソッドの処理内容が実装されていくのでしたね。では、今度は逆に、階層を下から上に昇ってみましょう。

以下の条件に沿って Monster クラスの親クラスを作っていきます。

① Monster と Character の共通の親として戦闘に参加する動物（BattleCreature）を定義します。戦闘に参加する動物の中には、専守防衛的な動物もいるかもしれませんので attack() は定義できません。

② BattleCreature の親として動物（Creature）を定義します。これは村人やお姫様のように戦闘に参加しない生き物も含んでいるため、HP フィールドはあるとは限りませんが、どのような動物であっても脅威から逃げるための run() は持っています。

Goblin であれば HP、MP、名前、攻撃力などのフィールドと、attack() や run()、useItem() などの内容が確定した具体的なメソッドを備えているでしょう。しかし Creature のようにあいまいになると、攻撃力や useItem() はおろか、名前さえも持っているとは限りません。**もはや「逃げる run() メソッドぐらいは最低でも持っている」ということしか決められない**のです。

この例に限らず、正しく継承が用いられている継承ツリーを上へ辿ると、次のような現象が順に起こります（次ページの図 12-15）。

①抽象メソッドが増える

「内容は確定できないが、一応存在する」という抽象メソッドが現れ始めます。

②抽象メソッドやフィールドが減っていく

クラスに定義してある抽象メソッドやフィールドが減っていきます。

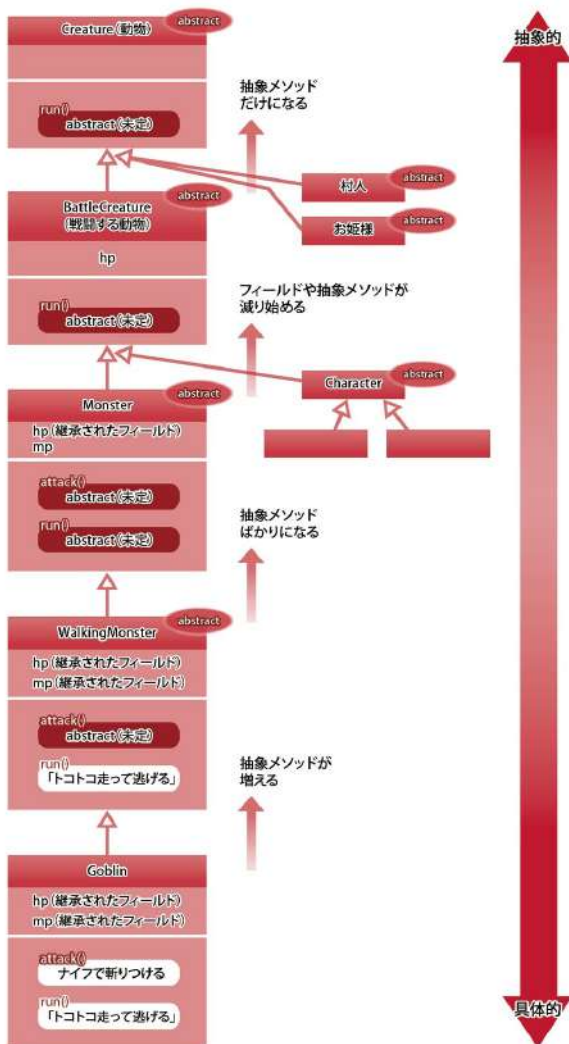



図 12-15　抽象継承階層を上に辿っていくと…

継承階層を上に辿るということは、「どんどんあいまいなものになっていく」ということです。クラスがあいまいになるにつれ、「どのような内部情報を持っているか(フィールド)」「どのような動きをするか(メソッド)」は、あやふやになり、決めることができなくなっていくのです。

12.4.2 抽象クラスの特別扱い

Creature クラスぐらいになると、ものすごくあいまいで「抽象クラスの中の抽象クラス」みたいな感じがしてきますね。

そうだね。そして、そんな「抽象クラスの中の抽象クラス」だけを特別扱いする文法が Java にはあるんだよ。

ここまで見てきたように、継承階層を上に辿ると、上流のクラスはすべて抽象クラスになります。そして Java では、次の条件を満たす「特に抽象度が高い抽象クラス」を、**インタフェース**(interface)として特別に扱うことができます。

インタフェースとして特別扱いできる 2 つの条件

①すべてのメソッドは抽象メソッドである。
②基本的にフィールドを 1 つも持たない。

たとえば次に示す抽象クラス Creature のコードを見てください。

リスト 12-11

Creature.java

```
1  public abstract class Creature {
2    public abstract void run();
3  }
```

このクラスには抽象メソッドしかなく、フィールドもありません。このまま抽象クラスとしておいてもよいのですが、以下のような構文を用いてインタフェースとして定義することも可能です。

インタフェースの宣言

アクセス修飾子 interface インタフェース名 {
⋮
}

では、インタフェースとして宣言したCreatureを見てみましょう。

リスト 12-12

Creature.java

```
1  public interface Creature {
2    public abstract void run();
3  }
```

なお、「インタフェースに宣言されたメソッドは、自動的にpublicかつabstractになる」というルールがあるので、通常は次のように書きます。

リスト 12-13

Creature.java

```
1  public interface Creature {
2    void run();
3  }
```

public abstractを省略しても大丈夫

初めてインタフェースという用語を聞いたときは、クラスとはまったく関係ない新しい何かだと思っていました。

でも、実は「クラスの仲間」で、「抽象クラスの親戚みたいなもの」なんですね。

そうだよ。初めのうちは難しく考えすぎないで、「あまりにあいまいすぎて特別扱いされた抽象クラス」と理解しておけばいいよ。

インタフェースにおける定数宣言

先ほどの「インタフェースとして特別扱いできる2つの条件」によれば、インタフェースは基本的にフィールドを持ちません。しかし、「public static final が付いたフィールド（定数）」だけは宣言が許されます。

さらにそのようなフィールドを宣言する場合は、「public static final」を省略してもよいことになっています。つまり、インタフェース内でフィールドを宣言すると自動的に public static final が補われ、定数を宣言したことになるのです。

次のコードは、円周率 PI をインタフェースに定義した例です。

Circle.java

```
1  public interface Circle {
2    double PI = 3.141592;
3  }
```

自動的に public static final が補われる

12.4.3 インタフェースの名前の由来

特にあいまいな抽象クラスを特別扱いするのはわかりましたけど、なぜ「インタフェース」という新しい名前なんですか？「スーパー抽象クラス」とかでいいんじゃないのかな？

なぜ２つの条件を満たした抽象クラスに「インタフェース」という、まったく新しい別の名前が付いているのでしょうか。その理由を探るため、次のインタフェースを見て意味を考えてみましょう。

リスト 12-14

CleaningService.java

```
1  public interface CleaningService {
2    Shirt washShirt(Shirt s);
3    Towl washTowl(Towl t);
4    Coat washCoat(Coat c);
5  }
```

このCleaningServiceはシャツとタオル、そしてコートを渡せば、それを洗って返してくれます。しかし布団やマフラーは扱っていないようですね。また、すべてのメソッドは抽象メソッドであり、処理内容が記述されていません。つまり、**「どのようにして洗うか」というクリーニング店の内部で行われる作業については明かされていない**わけです。このCleaningServiceインタフェースは、まるでクリーニング店の店頭メニューのようですね。

図 12-16　クリーニング屋のメニューには、どのように洗うかまでは書いていない

店頭メニューは、クリーニング店が「こういう仕事を受け付けますよ」と表明するためのものです。そしてお客さんはメニューを見て、「この仕事をお願いします」と依頼をします。

つまり、メニューは店主とお客さんとの接点（英語で「interface」といいます）の役割を果たしているのです。

12.4.4 インタフェースの実装

CleaningService インタフェースが店頭メニューだとすれば、それを継承して次のように記述したKyotoCleaningShop クラスこそが「クリーニング店そのもの」といえるでしょう。

リスト 12-15

KyotoCleaningShop.java

```
1  public class KyotoCleaningShop implements
   CleaningService {
2    private String ownerName;
3    private String address;
4    private String phone;
5    /* シャツを洗う */
6    public Shirt washShirt(Shirt s) {
7      // 大型洗濯機15分
8      // 業務用乾燥機30分
9      // スチームアイロン5分
10     return s;
11   }
12   /* タオルを洗う */
13   public Towl washTowl(Towl t) {
14     ：
15   }
16   /* コートを洗う */
17   public Coat washCoat(Coat c) {
18     ：
19   }
20 }
```

KyotoCleaningShop クラスの 1 行目にあるように、インタフェースを継承して子クラスを定義する場合は extends ではなく **implements** を使います。これを「CleaningService インタフェースを**実装して** KyotoCleaningShop を作る」などと表現します。

インタフェースの実装

アクセス修飾子 class クラス名 implements インタフェース名 {
　　⋮
}

なお、インタフェースという名前であっても、「しょせんは抽象クラスみたいなもの」ということを思い出してください。インタフェースで定義された washShirt()、washTowl()、washCoat() は、すべて抽象メソッドですので、子クラスである KyotoCleaningShop で、それぞれオーバーライドしなければなりません。

なお、図 12-17 で示したように、クラス図ではインタフェースの実装を点線の矢印記号で表します。

図 12-17　インタフェースの実装

実装する（implements）という用語が使われるのは、**親インタフェースで未定だった各メソッドの内容をオーバーライドして実装し確定させるから**だよ。

ところで、全国チェーンのクリーニング店では、どの店でも同じ店頭メニューを使っていることがあります。おそらく本社で作ったメニューを、すべての店で掲示しているのでしょう。

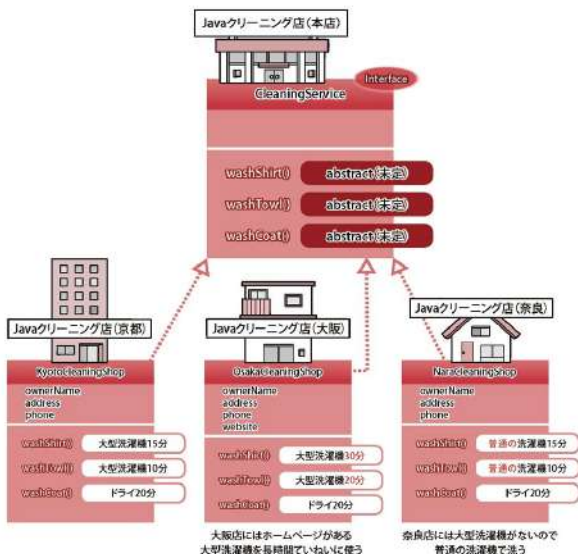



図 12-18　1 つのインタフェースを実装する複数のクラス

チェーン店とはいえ、京都店・大阪店・奈良店は別の店ですので、それぞれの店が持つ設備や洗濯の手順もさまざまでしょう。つまり、共通の CleaningService を実装していたとしても、個々の～ CleaningShop クラスのフィールドやメソッドの詳細は異なっても構いません。

しかし、このクリーニングチェーンの加盟店は、共通の店頭メニューを出している以上、どの店も「シャツ・タオル・コートの洗濯」はできる必要があります。個々の～ CleaningShop クラスは、CleaningService インタフェースで定められた抽象メソッドをオーバーライドして処理を実装している必要があるのです。

このように考えると、あるインタフェースに複数のメソッドを定義しておくことは、次のような 2 つの効果を生み出すと考えることができます。

インタフェースの効果

①同じインタフェースを **implements する複数の子クラスたちに、共通のメソッド群を実装するよう強制できる。**

②あるクラスがインタフェースを実装していれば、**少なくともそのインタフェースが定めたメソッドは持っていることが保証**される。

12.4.5 特別扱いされる理由

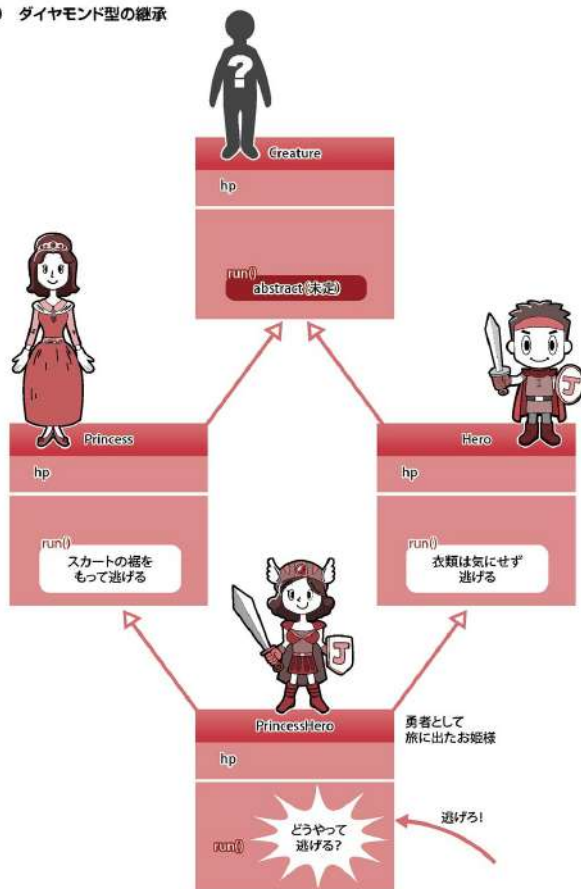
なるほど、インタフェースの名前の由来はわかりました。でも、なぜ2つの条件（12.4.2 項「インタフェースとして特別扱いできる2つの条件」参照）を満たす抽象クラスをわざわざ特別扱いするんですか？

それは、この2つの条件を満たすクラスは、ある特別なことを実現できるからなんだよ。

クリーニング店の例で見たように、インタフェースは「このようなメソッド群を持ち、このような引数を与えれば、このような結果を返す」という表面的な確約をするだけで、その**内部実装（メソッドの処理動作）をいっさい定めていません。**インタフェースが特別扱いされるのは、この「内部実装をいっさい定義しない」という性質があるからです。この性質のおかげで、**インタフェースでは特別に多重継承が許されています。**

多重継承は第 11 章（11.1.5 項）で少し触れたように、あるクラスを作成する際に2つの親クラスを使うことができる、とても便利な機能です。しかし、多重継承は誤用されやすく危険なので、Java では基本的に「クラスの多重継承」は禁止されました。

なぜ多重継承が危険で禁止されたのかを次の例で考えてみましょう。

図 12-19　ダイヤモンド型の継承

PrincessHero は、Princess と Hero の両方からメソッドを継承しています。このとき、PrincessHero の run() メソッドが呼び出されたら、このキャラクターはどのように逃げるのでしょうか？　「スカートの裾を持って」逃げるのでしょうか？　それとも「衣類を気にせず」逃げるのでしょうか？

このように、多重継承を用いると、「両方の親から**同じ名前でありながら異な**

る内容の2つのメソッドを継承してしまう」ことが起こりえるため、「お姫様としての逃げ方」と「勇者としての逃げ方」のどちらが動くべきなのか混乱を招いてしまいます。しかし、これがHeroインタフェースとPrincessインタフェースからの多重継承ならば、どうでしょうか？

PrincessHeroはPrincessとHeroの両方から抽象メソッドであるrun()を継承しますので、これを必ず**「勇者として旅に出たお姫様の独自の逃げ方」でオーバーライドすることになる**でしょう。

この場合、PrincessHeroのrun()が呼び出されても、先ほどの「クラスの多重継承」のような混乱は起こらずに、PrincessHeroでオーバーライドし定義されたrun()が動くのです。

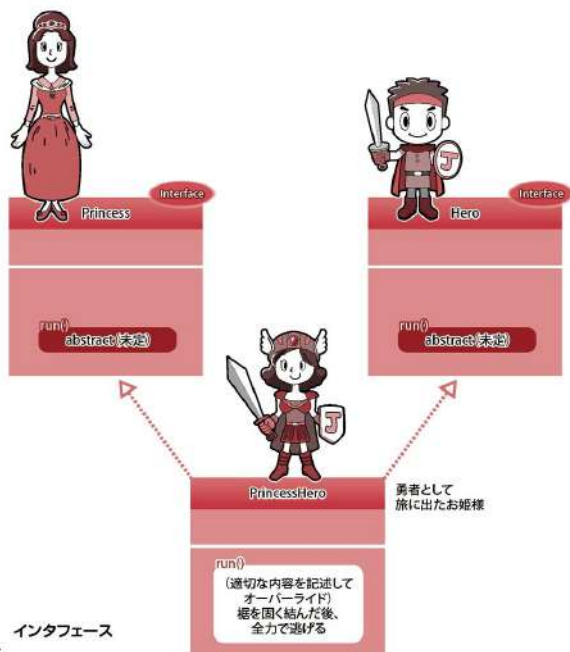



図12-20　インタフェースの多重継承

そもそも多重継承が問題なのは、「両方の親クラスから同じ名前でありながら**異なる内容**のメソッドを継承して衝突してしまう」からです。しかし、両方の親がインタフェースの場合、**どちらもメソッドの内容をいっさい定めていません**から、「親から継承した2つの処理内容が衝突する」ことは起こりえないのです。

クラスにはないインタフェースの特権

異なる実装が衝突する問題が発生しないため、複数の親インタフェースによる多重継承が認められている。

インタフェースによる多重継承は、次のような構文で行います。

インタフェースによる多重継承

アクセス修飾子 class クラス名
　　implements 親インタフェース名 1,
親インタフェース名 2,… {
　　⋮
}

次の例のように3つ以上のインタフェースを実装することも可能です。

PrincessHero.java

```
1  public class PrincessHero
       implements Hero, Princess, Character {
2    ⋮
3  }
```

これら3つはすべてインタフェースとします

12.4.6 インタフェースの継承

ところで、あるインタフェースを定義する場合、ゼロから開発せずに既存のインタフェースを拡張することもできます。

リスト 12-16

Human.java

```
1 public interface Human extends Creature {
2   void talk();
3   void watch();
4   void hear();
5   // さらに、親インタフェースからrun()を継承
6 }
```

あれ？ インタフェースを継承するときは extends ではなく implements を使うはずじゃ…。

今回の場合、Human は Creature の run() メソッドをオーバーライドして**処理の内容を確定しているわけではありません**ので、implements ではなく extends を使います。混乱しやすい部分かもしれません。

implements と extends の使い分けは次ページの表 12-1 のとおりです。

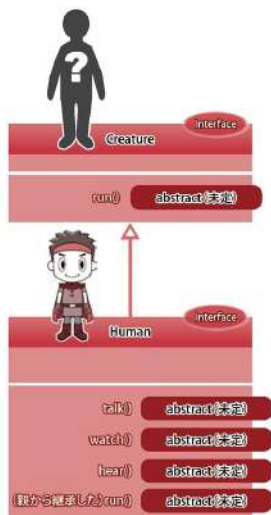



図 12-21　インタフェースの拡張

表 12-1　implements と extends の使い分け

継承元	継承先	使用するキーワード	継承元の数
クラス	クラス	extends	1つ
インタフェース	クラス	implements	1つ以上
インタフェース	インタフェース	extends	1つ以上

「同種（クラス同士、インタフェース同士）の継承の場合は extends を使う」「異種なら implements」と覚えておこう。

12.4.7 extends と implements を一緒に使う

クラス定義の際に extends と implements の両方を利用することもあります。

extends と implements の両方を使ったクラス定義

```
アクセス修飾子 class クラス名 extends 親クラス
    implements 親インタフェース1, 親インタフェース2,
…{
  ⋮
}
```


たとえば次のような使い方をします（リスト 12-17 および図 12-22）。

リスト 12-17

Fool.java

```
1 public class Fool extends Character implements Human {
2   // Characterからhpやget Name()などのメンバを継承している
3   // Characterから継承した抽象メソッドattack()を実装
4   public void attack(Matango m) {
```

```
5     System.out.println
          (this.getName() + "は、戦わず遊んでいる。");
6   }
7   // さらにHumanから継承した4つの抽象メソッドを実装
8   public void talk() { … }
9   public void watch() { … }
10  public void hear() { … }
11  public void run() { … }
12 }
```

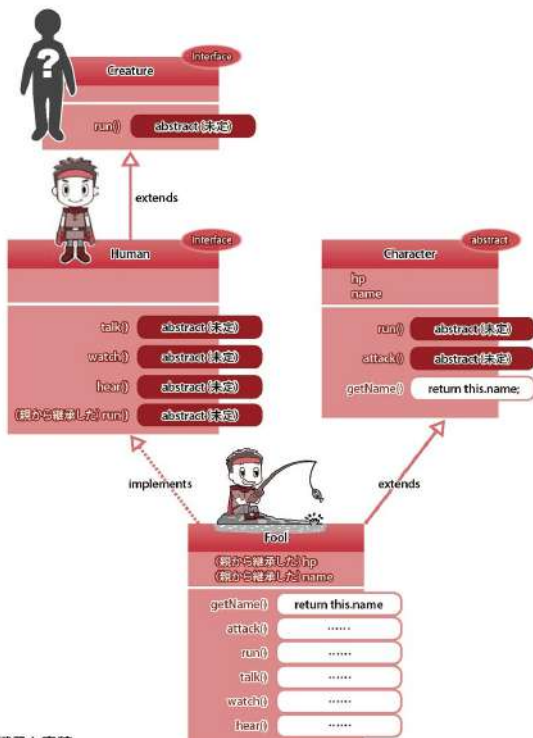



図 12-22　継承と実装の組み合わせ

インタフェースメソッドのデフォルト実装

12.4.1 項で紹介したように、インタフェースが持つことのできるメソッドは処理内容を持たない抽象メソッドに限られます。しかし Java8 以降では、default キーワードを用いることで、処理のデフォルト実装を添えた抽象メソッドを定義できるようになりました。

デフォルト実装付き抽象メソッドの宣言

```
default 戻り値 メソッド名 ( 引数 ) {
  処理のデフォルト実装
}
```

このようにして定義された抽象メソッドは、もし継承先でオーバーライドされなかった場合、自動的に、デフォルト実装として定めた処理内容でオーバーライドされたものと見なされます。

上手に使うとオーバーライドの手間を省くことができる便利な機能ですが、多重継承によるデフォルト実装の衝突を招くこともある点には注意が必要です。

第12章のまとめ

この章では、次のようなことを学びました。

継承の材料を作る開発者の立場と役割

- 「他の人が継承の材料として使うであろう親クラスを作る立場」の開発者も存在する。
- 「未来の開発者が効率よく安心して利用できる継承の材料を作ること」がその使命。
- その使命を達成するために、Java では抽象クラスやインタフェースという道具を提供している。

抽象クラス

- 中身を決定できない「詳細未定メソッド」には abstract を付けて抽象メソッドとする。
- 抽象メソッドを１つでも含むクラスは、abstract を付けた抽象クラスにしなければならない。
- 抽象クラスはインスタンス化することが禁止されている。
- 抽象クラスと抽象メソッドを活用した「継承の材料」としての親クラスを開発すれば、予期しないインスタンス化やオーバーライド忘れの心配がない。

インタフェース

- 抽象クラスのうち、基本的に抽象メソッドしか持たないものを「インタフェース」として特別扱いできる 。
- インタフェースに宣言されたメソッドは自動的に public abstract となり、フィールドは public static final になる。
- 複数のインタフェースを親とする多重継承が許されている。
- インタフェースを親に持つ子クラスの定義には implements を用いる。

問題12-1

ある会社では、会社の資産として保有するものを管理するプログラムを作ろうとしています。現時点では、「コンピュータ」「本」を表す、次のような２つのクラスがあります。

Book.java

```
1  public class Book {
2    private String name;
3    private int price;
4    private String color;
5    private String isbn;
6    // コンストラクタ
7    public Book
         (String name, int price, String color, String isbn) {
8      this.name = name;
9      this.price = price;
10     this.color = color;
11     this.isbn = isbn;
12   }
13   // getterメソッド
14   public String getName() { return this.name; }
15   public int getPrice() { return this.price; }
16   public String getColor() { return this.color; }
17   public String getIsbn() { return this.isbn; }
18 }
```

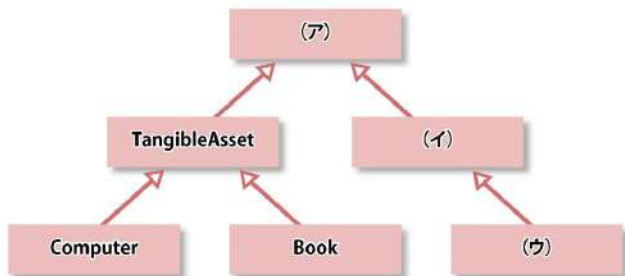
12章

Computer.java

```
public class Computer {
  private String name;
  private int price;
  private String color;
  private String makerName;
  // コンストラクタ
  public Computer
      (String name, int price, String color, String makerName) {
    this.name = name;
    this.price = price;
    this.color = color;
    this.makerName = makerName;
  }
  // getterメソッド
  public String getName() { return this.name; }
  public int getPrice() { return this.price; }
  public String getColor() { return this.color; }
  public String getMakerName() { return this.makerName; }
}
```

今後、コンピュータと本以外にも、さまざまな形ある資産を管理していきたい場合に有用な「有形資産（TangibleAsset）」という名前の抽象クラス（継承の材料）を作成してください。また、Computer や Book は、その親クラスを用いた形に修正してください。

問題 12-2

問題 12-1 の会社では、形のない無形資産（IntangibleAsset）も管理しようと考えています。無形資産には、たとえば特許権（Patent）などがあります。無形資産も有形資産も資産（Asset）の一種です。この前提に従い、次の継承図の（ア）～（ウ）にあてはまるクラス名を考えてください。

また、(ア)に入る抽象クラスを開発し、このクラスを継承するようにTangibleAssetを修正してください。

問題 12-3

資産かどうかとは関わりなく、形がある物(Thing)であれば、「重さ」を得ることができるはずです。そこで、double型で重さを取得するgetterメソッドgetWeight()とsetterメソッドsetWeight()を持つインタフェースThingを定義してください。

問題 12-4

有形資産(TangibleAsset)は、資産(Asset)の一種でもありますし、形ある物(Thing)の一種でもあります。この定義に沿うようにTangibleAssetのソースコードを修正してください。この際、TangibleAssetにフィールドやメソッドの追加が必要であれば、適宜追加してください。

12.7 練習問題の解答

問題 12-1 の解答

クラスの宣言に関する問題です。正解のコードは以下のとおりです。

TangibleAsset.java

```
1  public abstract class TangibleAsset {
2    private String name;
3    private int price;
4    private String color;
5    public TangibleAsset(String name, int price, String color) {
6      this.name = name;
7      this.price = price;
8      this.color = color;
9    }
10   public String getName() { return this.name; }
11   public int getPrice() { return this.price; }
12   public String getColor() { return this.color; }
13 }
```

Book.java

```
1  public class Book extends TangibleAsset {
2    private String isbn;
3    public Book
         (String name, int price, String color, String isbn) {
4      super(name, price, color);
5      this.isbn = isbn;
6    }
7    public String getIsbn() { return this.isbn; }
8  }
```

Computer.java
```
1  public class Computer extends TangibleAsset {
2    private String makerName;
3    public Computer
         (String name, int price, String color, String makerName ) {
4      super(name, price, color);
5      this.makerName = makerName;
6    }
7    public String getMakerName() { return this.makerName; }
8  }
```

練習12-2の解答

（ア）Asset　（イ）IntangibleAsset　（ウ）Patent

Asset.java
```
1  public abstract class Asset {
2    private String name;
3    private int price;
4    public Asset(String name, int price) {
5      this.name = name;
6      this.price = price;
7    }
8    public String getName() { return this.name; }
9    public int getPrice() { return this.price; }
10 }
```

TangibleAsset.java
```
1  public abstract class TangibleAsset extends Asset {
2    private String color;
3    public TangibleAsset(String name, int price, String color) {
4      super(name, price);
5      this.color = color;
6    }
7    public String getColor() { return this.color; }
```

```
8 }
```

練習12-3の解答

メソッドの宣言に関する問題です。以下は解答例のコードです（おおむね合っていれば正解とします）。

Thing.java

```
1 public interface Thing {
2   double getWeight();
3   void setWeight(double weight);
4 }
```

練習12-4の解答

以下は解答例のコードです（おおむね合っていれば正解とします）。

TangibleAsset.java

```
1  public abstract class TangibleAsset extends Asset
       implements Thing {
2    private String color;
3    private double weight;
4    public TangibleAsset(String name, int price, String color) {
5      super(name, price);
6      this.color = color;
7    }
8    public String getColor() { return this.color; }
9    public double getWeight() { return this.weight; }
10   public void setWeight(double weight){ this.weight = weight; }
11 }
```

第 13 章

多態性

前章では抽象クラスやインタフェースを用いて、現実世界にある
あいまいな物事を Java のコードとして表現できることを学びました。
私たちは日常生活でも、無意識に物事を
「あいまいに捉える」「あいまいに使う」ことにより、
さまざまなメリットを享受しています。
本章で学ぶ最後のオブジェクト指向の 3 大要素の 1 つ「多態性」とは、
Java 仮想世界でも物事を「あいまいに捉える」ための機能です。

CONTENTS

13.1 多態性とは

13.1.1 開発をラクにする多態性

多態性（polymorphism）はオブジェクト指向プログラミングを支える３大機能の１つで、多様性やポリモーフィズムと呼ばれることもあります。

また、難しそうな名前だなぁ…。

だけど、この多態性をマスターすると、湊くんの RPG 作りは何倍もラクになるんだよ。

そもそもオブジェクト指向は、「ラクしてよいものを実現する」ためのものでした。特に、**多態性を上手に活用すると、とても効率よく楽しく開発できる**と聞けば、がんばって覚えようという気になりませんか？

カプセル化や継承と同じく、多態性の学習にも「コツ」があります。実は、多態性を定義や文法規則から学び始めると、大変難しく感じます。しかし、イメージを理解してから文法や定義を学べばそれほど難しいものではありません。

この章では、多くのイメージ図を示しながら、よりやさしくマスターできるように解説を進めていきます。ぜひ頭の中にイメージを広げながら、気楽に読み進めてください。

13.1.2 多態性のイメージ

多態性の定義は章の最後に紹介します。また、「多態」の意味についても、今は考えないでください。現時点では、次のような理解で十分です。

多態性のあいまいなイメージ

「あるものを、あえてザックリ捉える」ことで、さまざまなメリットを享受しようという機能。

この章を学ぶにあたっての大事なキーワードは「ザックリ」です。ザックリ捉えるとは、たとえば以下のような考え方です。

厳密に言えば SuperHero なんだけど、まぁザックリいえば Hero だよね。
厳密に言えば GreatWizard なんだけど、まぁザックリいえば Wizard だよね。
厳密に言えば Slime なんだけど、まぁザックリいえば Monster だよね。

このような捉え方をして、さまざまなメリットを享受しようというのが多態性という機能なのです。

13.1.3 ザックリ捉えるメリット

ザックリ捉える、というのはなんとなくわかりましたけど、そんなことでメリットなんてあるんですか？

もちろんさ。ザックリなしでは、人間は生きていけないよ。

13章

ザックリ捉えることによるメリットは、私たちの日常生活にも多く見ることができます。たとえば、レンタカーを借りて車を運転するときのことを考えてみましょう。厳密にいえば初めて乗る車であるにも関わらず、多くの人は問題なく運転できます。なぜ、初めての車なのに運転できるのかとドライバーに聞けば、おそらく次のような答えが返ってくるでしょう。

「まぁハンドルは同じだし、右ペダルがアクセル、左がブレーキ。
細かいところはあれこれ違うけど、まぁザックリみれば、どの車も同じだよ。」

この人は高級車や軽自動車、ライトバン、さらには来年発売の新車（現時点では未知の車）でも問題なく運転できるでしょう。

「そんなの当たり前じゃないか」と思うかもしれませんが、もし運転するのがロボットだとしたら、こうはいきません。ロボットに内蔵される運転制御プログラムは、たとえば以下のように無数の細かい設定が必要と考えられます。

> もし 1997 年式のプリウスなら、ハンドルはシートから○ cm の高さにあり、それを 10 度回すとタイヤが○度曲がり…。
> もし 1998 年式のインプレッサなら、…（以下、延々と続く）

ロボットは、それぞれの車種について細かい条件を完全に把握している必要があり、かつ「把握してない車」は操作できません。

明らかに人間のほうが「ラク」をして同じ成果が得られていますが、それは、**車の厳密な車種についてはあまり考えていない＝ザックリと「車」としか捉えていない**からです。

車に限らず、私たち人間は、世の中にあるさまざまな複雑なものをザックリ捉えることによって、厳密には違うものも「似たようなもの」として上手に利用しています。この「私たちが現実世界でラクするために用いている方法」をプログラムでも実現する機能こそ、オブジェクト指向３大機能における最後の１つ、**「多態性」**なのです。

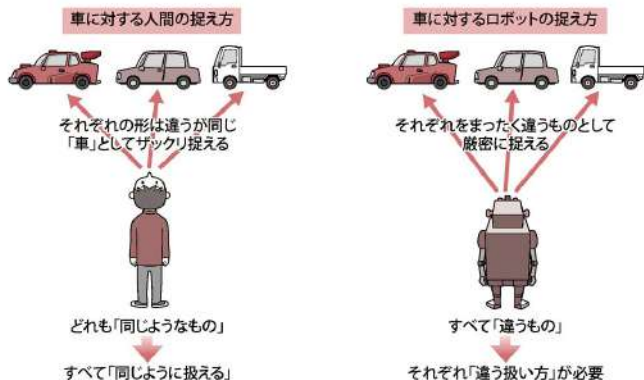



図 13-1　ザックリ捉える人間、厳密に捉えるロボット

13.2 ザックリ捉える方法

13.2.1 ザックリ捉えるための文法

確かに、Java の世界でも「ザックリ」を実践できたら便利そうね。

ザックリ捉えるには、どんな文法やキーワードを使えばいいんですか？

実はザックリ捉えるための特別な文法はないんだよ。

今まで学習したオブジェクト指向の機能には、特有のキーワードが登場しました。カプセル化といえば「private」や「public」、継承といえば「extends」がすぐに思い浮かぶでしょう。

そこで多態性も**何か専用の文法があって、それを書けば利用できると思い込んでしまう**人がいますが、**多態性には専用の文法はありません。**

実は、今まで何度も用いてきた「代入の文法」を使えば、ザックリ捉えることができるようになっています。

それではさっそく、SuperHero クラスを用いて解説していきましょう。まず前提として SuperHero は親として Hero クラスを、さらにその親として Character クラスを持っています（p.437 の図 11-14）。

通常、SuperHero のインスタンスを生成して利用するには、次のような文を記述するのでしたね。

```
SuperHero h = new SuperHero();
```

このときの状態をイメージで表すと、次ページの図13-2のようになります。ここで、箱に書かれた「<<SuperHeroです>>」という文字は、変数hの型を表しています。

SuperHero型の変数にSuperHeroのインスタンスが入っているわけですね。通常、変数にはその型と同じ型の内容が入りますから、これはごく当たり前の状態です。

図13-2　SuperHero型の変数hに格納されたSuperHeroインスタンス

次に、SuperHeroを「ザックリCharacterとして捉える」書き方です。

```
Character c = new SuperHero();
```

第8章では「newをするときは左辺と右辺に同じ型を書く」と紹介しました（8.4.2項）が、このように左辺と右辺の型を変えることも実は可能です。今回の場合、左辺の型がCharacterになっただけですが、イメージ図は図13-3のように変わります。

図13-3　Character型の変数cに格納されたSuperHeroインスタンス

箱の中身のインスタンスは正真正銘のSuperHeroですが、箱の表面には「Characterです」と書かれています。よって、以後この変数cを利用するときは、**本当はSuperHeroインスタンス**なのですが、**あくまでCharacterとして捉えて利用する**ことになります。

このように、多態性を活用するためには「箱の型」と「中身の型」という異なる2つの型が関係してきます。そして**あるインスタンスをどのように捉えるかは、どの型の変数に代入するか（箱の型）で決まります。**

Character c = new SuperHero();

箱の型

そのインスタンスを「何と見なす」か。同じSuperHeroインスタンスでもCharacter型やHero型など、さまざまな箱に入れ替えることで、捉え方を変えることができる

中身の型

そのインスタンスが、いったい「何」かは、一度newされたら何があっても変わることはない

図 13-4　箱の型と中身の型

13.2.2 できる代入、できない代入

new をするときの左辺と右辺は、別に同じ型でなくてもいいんですね。

そうだよ。ただし、どんな型でもいいわけではないんだ。

次のコードを見てください。1 行目はエラーになりませんが、2 ～ 4 行目はすべてエラーになります。

```
Character c = new SuperHero();    // OK!
Sword s = new Hero();             // エラー
Flower f = new Fish();            // エラー
Phone p = new Coffee();           // エラー
```

第 1 章の 1.3.1 項で解説したように、代入式は基本的に「左辺の型と右辺の型が異なる場合はエラー」になります。たとえば「String str = 1;」がエラーになるのは当然ですね。しかし、この原則でいえば 1 行目のコードも「左辺は Character 型、

右辺はSuperHero型」なのでエラーになるはずです。なぜ1行目だけがエラーにならず、特例として許されているのでしょうか？

その「代入が許される判断基準」は絵を描いてみればわかります。先ほどの図13-3をもう一度見てください。

箱には「Characterです」と書いてあり、中身にはSuperHeroが入っています。そしてこの**絵の内容は（厳密ではありませんが）嘘ではありません**。スーパーヒーローもキャラクターの一種（SuperHero is-a Character）ですから、「キャラクターが入っています」という箱に入っていても別に矛盾はないのです。

Javaではこのように、**絵に描いてみて嘘にならないインスタンスの代入は許されます**。

しかし、次のような代入はエラーになります。

```
float f = new Hero();
Item i = new Hero();
SuperHero sh = new Hero();
```

図13-5　代入ができない例

絵で描いてみればわかるように、これらはいずれも嘘になってしまうからです。

よく「子クラスのインスタンスは親クラスの型に代入可能」などという定義を丸暗記しようとして混乱する人がいる。だが、「イメージ図を描いてみる」ほうが忘れにくいしオススメだよ。

やっぱりオブジェクト指向ってイメージが大事なんですね。

13.2.3 継承のもう1つの役割

前節で「絵に描いて嘘がないならば代入可能」なことはわかりました。ただし1つだけ注意点があります。

そもそも「絵に嘘が含まれるか」を判断するには、「〜は〜の一種である」という前提知識が必要です。たとえば私たちは「ヒーローの中で特に選ばれた者がスーパーヒーローである（つまり、スーパーヒーローはヒーローの一種である）」という前提知識があるからこそ、図13-3が「嘘ではない」と判断できました。

私たち人間には「常識」がありますので、「魔術師が生き物の一種であること」や「剣が武器の一種であること」を知っています。しかし、Javaには「何が何の一種か」という一般常識は備わっていません。

そこでJavaは、extendsやimplementsを用いた継承関係にあるクラス同士について、「片方のクラスは、他方のクラスの一種（is-aの関係）」であると考えます。言い換えれば、**extendsやimplementsはプログラマが「is-aの関係」をJavaに伝える手段**でもあるのです。

そのため、いくら私たち人間が考えた一般常識ではis-aの関係であっても、Javaの継承関係で2つのクラスがつながっていなければ代入はできません。

常識では「ヒーローは人間の一種」
だが、クラス同士は無関係だとすると…

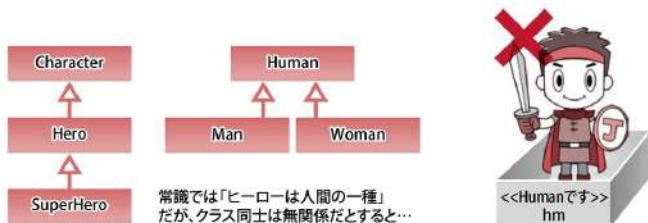



図13-6　extendsでis-a関係が結ばれていないと代入できない

第11章の最後でも触れたけど、「is-aの関係（抽象的・具体的の関係）をJavaに知らせる」というのも継承の重要な役割なんだ。

だから「仮にラクをできるとしても、is-a の関係でないなら継承を使ってはならない」と教えられたわけですね（11.4.2 項）。

13.2.4　箱の型に抽象クラスを使う

ここで、あなたが命あるあらゆるものの親として、新たに Life インタフェースを作ったと考えましょう。Character も Life を実装するように変更したと仮定します。第 12 章で「抽象クラスやインタフェースはインスタンス化できない」と学びました。ですからインタフェース Life が定義されている場合、「new Life()」とすることはできません。

しかし、第 8 章で学んだ「クラス定義によって可能になる 2 つのこと（8.3.1 項）」を思い出してください。インタフェースである Life はインスタンス生成のためには利用できませんが、変数の型（表面に「Life が入っています」と書かれた箱）として使うことはできます。たとえばリスト 13-1 のように、Life 型の変数に Wizard インスタンスを入れることができます。

リスト 13-1

Life.java

```
1 public interface Life { … }
```

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Life lf = new Wizard();
4   }
5 }
```

Wizard は生き物の一種

抽象クラスやインタフェースの型

抽象クラスやインタフェースからインスタンスを生み出すことはできないが、型を利用することは可能。

ザックリ捉えたものに命令を送る

13.3.1 捉え方の違いは使い方の違い

前節では、どのようにして「ザックリ捉えるか」を紹介しました。この節では、「あるインスタンスをザックリ捉えるのと、厳密に捉えるのでは、その利用にどのような違いが出てくるか」を見ていきます。

多態性の説明を一休みして、1つたとえ話をしましょう。

> この紙切れを原始人に渡すと「絵」と捉えるでしょう。そして原始人は、これを見て楽しむことはあっても、何かに使えるとは思わないでしょう。
>
> 一方、同じものを湊くんに渡すと、それはもう大喜びします。絵の人物が「福沢諭吉」であり、その紙切れが厳密には「紙幣」だということを知っているからです。彼はその紙切れを眺めることもできますが、どこかの店に持ち込み、何か商品と交換してしまうでしょう。

図 13-7　何と捉えるかによって、用途が増える

このたとえ話が示唆しているのは、**「まったく同一である1つの存在に対して、複数の異なる捉え方ができる」**ということと、**「何かを利用する人は、それを何と捉えているかによって、利用方法が変わる」**ということです。あいまいで抽象的なほど用途は限定され、具体的に捉えるほど用途が増えていきます。

13.3.2 呼び出せるメソッドの変化

この「捉え方が変わると、利用方法が変わる」という現実世界の現象は、実はJava仮想世界でもちゃんと再現されています。p.437の図11-14におけるCharacterとWizardを使って、このことを解説していきましょう。まず、次の2つのクラスのソースコードを見てください。

リスト13-2

Character.java

```
public abstract class Character {
  String name;
  int hp;
  public abstract void attack(Matango m);
  public void run() { … }
}
```

Wizard.java

```
public class Wizard extends Character {
  int mp;
  public void attack(Matango m) {
    System.out.println(this.name + "の攻撃！");
    System.out.println("敵に3ポイントのダメージ");
    m.hp -= 3;
  }
  public void fireball(Matango m) {
    System.out.println(this.name + "は火の玉を放った！");
    System.out.println("敵に20ポイントのダメージ");
    m.hp -= 20;
    this.mp -= 5;
  }
}
```

Wizard は魔法使いとして attack() や fireball() のメソッドを持っていますので、インスタンス化すれば attack させたり fireball を使わせたりできます。

リスト 13-3

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Wizard w = new Wizard();
4     Matango m = new Matango();
5     w.name = "アサカ";
6     w.attack(m);
7     w.fireball(m);
8   }
9 }
```

さて、13.2 節で学んだように Wizard は Character の一種なので、Character 型変数に代入することが可能です。しかし、いざ Character 型に代入して fireball を呼び出そうとするとエラーが起きます。

リスト 13-4

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Wizard w = new Wizard();
4      Character c = w;          ← Character 型の箱に代入
5      Matango m = new Matango();
6      c.name = "アサカ";
7      c.attack(m);
8      c.fireball(m);            ← この行でエラーが発生する
9    }
10 }
```

え？ いくら Character 型の箱に入っているとはいえ、中身は正真正銘の魔法使いさんのはずなのに…。

魔法使いであれば fireball() を呼び出せるはずなのに、どうしてエラーになるんだろう？

それはね、「本当は魔法使いなんだけど、呼び出す側が魔法使いと思ってない」から呼び出せないんだよ。

Character 型の変数 c に格納されているとはいえ、箱の中身のインスタンスは正真正銘の Wizard インスタンスです。そして Wizard ならば fireball が使えるはずです。それにも関わらず、なぜコンパイルエラーになるのでしょうか？

この章の「ザックリ捉える文法（13.2.1 項）」で説明したとおり、Character 型の変数に代入するということは、中身のインスタンスを「（Hero だか Wizard だかわからないけど）なにかのキャラクター」程度にザックリ捉えるということにほかなりません。よって、**箱の中身が Wizard であることを忘れてしまいます**。

あいまいな型の箱へのインスタンスの代入

インスタンスをあいまいに捉えることとなり、「厳密には何型のインスタンスだったか」がわからなくなってしまう。

リスト 13-4 の 3 行目では確かに魔法使いを生み出しています。しかし 4 行目の代入を行った瞬間、私たちは箱 c の中身が「Hero なのか Wizard なのか、はたまた別の職業のクラスなのか」がわからなくなってしまいます。**確実に言えることは、「この箱に入っているのは、キャラクターの一種であること」だけ**です。

そう考えると、attack() が呼び出せて fireball() が呼び出せなかった理由にも説明がつきます。

■ attack() が呼び出せた理由

箱の中身が Hero でも Wizard でも、Character の一種である限り attack() メソッドは継承して持っているはずだから（どんなキャラクターでも最低限、攻撃はできるはずだから）。

■ fireball() が呼び出せなかった理由

箱の中身が Hero の場合など、fireball() メソッドを持っている職業とは限らないから（キャラクターであれば必ず火の玉を放てるとは限らないから）。

図 13-8　本当は Wizard だが、箱の外部からはわからない

箱の中の魔法使いさんが火の玉を放てなくなっちゃったわけではないんですね。

彼自身は「火の玉を放って」とお願いされれば、いつでも放つことができるんだ。ただ、私たちが彼を「魔法使い」と認識しないと「お願いできない」（しようと思わない）だけなんだよ。

私たちがこのインスタンスを「Character」と捉えている限り（= Character 型の変数に入っている限り）、私たちはこのインスタンスに「少なくとも Character ならできる最低限のこと」しか命令できません。箱の中身のインスタンスがどんなに多くのメソッドを持っていたとしても、Character として持つメソッドだけしか外部からは**呼び出すことができない**のです。

13.3.3 メソッドを呼び出せた場合に動く処理

前項では「どの箱に入れるかによって、呼べるメソッドが変わる」ことを学習しました。では次に、「もし**メソッドが呼べたとしたら**、その動きはどうなるか」についてモンスター関連クラスを題材に実験してみましょう。

リスト 13-5

Monster.java

```
public abstract class Monster {
  public void run() {
    System.out.println("モンスターは逃げ出した。");
  }
}
```

Slime.java

```
public class Slime extends Monster {
  public void run() {
    System.out.println("スライムはサササっと逃げ出した。");
  }
}
```

Main.java

```
public class Main {
  public static void main(String[] args) {
    Slime s = new Slime(); Monster m = new Slime();
    s.run(); m.run();
  }
}
```

さぁ問題だ。実行結果として、画面には何と表示されるかな？

Slime型のrun()とMonster型のrun()を呼び出しているってことは…。

まず「スライムはサササっと逃げ出した。」、次に「モンスターは逃げ出した。」かな。

実行結果

```
スライムはサササっと逃げ出した。
スライムはサササっと逃げ出した。
```

s.run() の結果

m.run() の結果

実行結果の２行目に注目してください。Monster 型の変数 m の run() メソッドを呼び出しているのに「モンスターは逃げ出した。」という表示になりません。

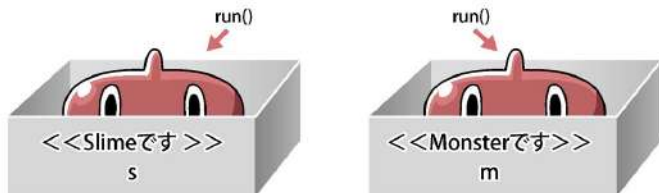



図 13-9　逃げ出すのは「実体」

変数 s と変数 m とは、箱の表面に書かれた「○○が入っています」という表記に違いがあるものの、両方とも中身はあくまでスライムです。ですから「逃げろ！」という命令が届きさえすれば、当然**スライムが**逃げます。つまり、Slime の run() が動作するのです。

このように、実際に動くメソッドの中身はインスタンスの型（中身の型）によって決まります。それが**どんな型の箱に入っているかは関係ありません**。

最後に「箱の型」と「中身の型」について、もう一度まとめておきましょう。

「箱の型」と「中身の型」

[箱の型]　どのメソッドを「呼べるか」を決定する。

[中身の型] メソッドが呼ばれたら、「どう動くか」を決定する。

捉え方を変更する方法

13.4.1 捉え方を途中で変える

図 13-8（13.3.2 項）を見て思ったんだけど、Character 型の箱に入れられた魔法使いに対しては、もう 2 度と fireball() を呼び出せなくなっちゃうのかなぁ？

彼を「魔法使いだ」と捉え直せば、またお願いできると思うけど…。

次のコードを見てください。

```
Character c = new Wizard();
```

このとき、変数 c に対して fireball() メソッドを呼べなくなることはすでに学びました。しかし、「中身が本当は Wizard だとわかっているし、どうしてもこのインスタンスに fireball を使わせたい」という場合がまれにあります。

fireball を使えるようにするためには、変数 c の中身を「**Wizard であると捉え直す**」必要があります。そのためにはインスタンスを Wizard 型変数に代入すればよいと想像がつくでしょう。しかし、次のコードはエラーになります。

```
1  Character c = new Wizard ();
2  Wizard w = c;        惜しい！エラーになる
```

なぜコンパイラは、この代入を許さないのでしょうか？ 私たち開発者はコード 1 行目の経緯を知っています。そのため、「もともと Wizard インスタンスとして生み出して、それを Character と見なしたものの、再び元の Wizard と見直

して何が悪い！」と感じます。

しかしコンパイラは基本的に**プログラムを1行ずつ解釈・翻訳しようとします**ので、今回のエラーも2行目だけを見て出しているのです。

2行目だけを見ると論理上はCharacter型の変数cに入っている中身は、HeroやThiefの可能性もあります。万が一、変数cの中身がHeroなのに、それをwに代入してしまったら、「Wizard型の変数にHeroインスタンスが入っている」という嘘の構図になってしまいます。

ですからコンパイラは、このような失敗する**可能性のある**代入について、「中に入っているものが常にWizardとは限らないから代入できない」と拒否します。

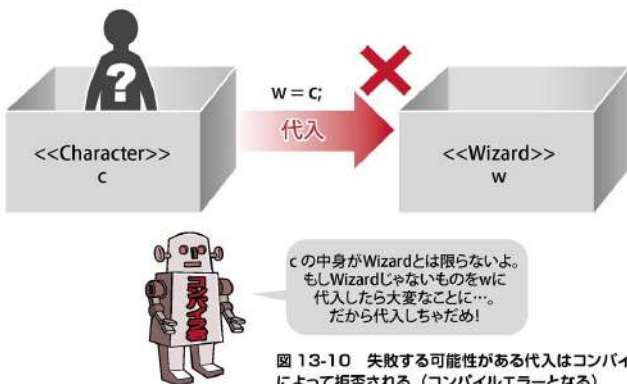



図13-10　失敗する可能性がある代入はコンパイラによって拒否される（コンパイルエラーとなる）

それでも変数cの中身を強制的に「Wizardとして捉え直したい」場合には、次のように書きます。

```
1  Character c = new Wizard();
2  Wizard w = (Wizard) c;
```


第2章で登場した「キャスト演算子」の再登場です。この演算子は強制的な型変換をコンパイラに対して明示的に指示する、とても強力な演算子です。こうすればコンパイラは文句を言わず、コンパイルを通してくれます。特に、今回のような「あいまいな型に入っている中身を厳密な型に代入する」キャストは**ダウンキャスト**（down cast）といわれ、失敗の危険が伴います。

13.4.2 キャストの失敗

ダウンキャストは「代入が失敗する可能性」を懸念してエラーを出すコンパイラに対し、「代入をしても矛盾のある状態にはならないから、黙って代入しろ」と頭ごなしに型変換を実行させる命令です。前節のコードの例ではうまく動きますが、次のようなケースではどうでしょうか？

```
1  Character c = new Wizard();
2  Hero h = (Hero) c;
```

いいから黙って Hero と見なせ！

このソースのコンパイルは成功します。しかし動作させると、実際に代入しようとした瞬間に ClassCastException というエラーが発生します。このエラーは、「キャストによる強制代入の結果『嘘の構図』になったので強制停止せざるを得なくなった」という意味のエラーです。

図 13-11　誤ったキャストにより ClassCastException が発生する

13.4.3 インスタンスを代入可能かチェックする

ダウンキャストによるClassCastExceptionを確実に回避するには、キャストする前に「キャストしても大丈夫かどうか」をチェックします。Javaにはそのための**instanceof演算子**が用意されています。

安全にキャストできるかを判定する

変数 instanceof 型名

※変数を型名の箱に代入可能ならばtrueが返る。

instanceof演算子は、「指定の型に代入しても絵として嘘にならないか」を判定してくれます。この演算子を利用して、たとえば次のようなコードを記述することもできます。

```
1  if (c instanceof SuperHero) {
2
3    SuperHero h = (SuperHero) c;
4    h.fly();
5  }
```

もしcの中身がSuperHeroと見なして大丈夫なら…

「SuperHero」と見なせ！

13章

13.5 多態性のメリット

13.5.1 ザックリ捉えることによるメリット

ザックリ捉えると、呼び出せるメソッドが減ってしまうんですよね？ それって不便じゃないですか？

呼び出せるメソッドが減っても、ザックリ捉えることでそれに勝るメリットがあるんだ。

13.2 節では、「多態性を用いて、インスタンスをザックリ捉える方法」を学びました。そして 13.3 節では、それによって「呼び出せるメソッドが減る」ことも学びました。

これだけを見ると、「ザックリ捉えることは、利用できるメソッドが減るだけで、まったくメリットがない」ように感じられますが必ずしもそうではありません。この節では、多態性を用いてザックリ捉えることによるメリットを具体的に Java のコードで紹介していきます。

13.5.2 同一視して配列を利用する

5 人のキャラクター(Hero が 2 人、Thief が 1 人、Wizard が 2 人)が旅をするゲームを考えてみましょう。この 5 人は 1 つのパーティ(冒険のためのチーム)です。彼らが宿屋に泊まり、全員の HP を 50 ずつ回復するプログラムを書く場合、次のようなものになるでしょう。

リスト 13-6

Main.java

```
1 public class Main {
```

```
2    public static void main(String[] args) {
3      Hero h1 = new Hero();
4      Hero h2 = new Hero();
5      Thief t1 = new Thief();
6      Wizard w1 = new Wizard();
7      Wizard w2 = new Wizard();
8      // 冒険開始！
9      // まず宿屋に泊まる
10     h1.setHp(h1.getHp() + 50);
11     h2.setHp(h2.getHp() + 50);
12     t1.setHp(t1.getHp() + 50);
13     w1.setHp(w1.getHp() + 50);
14     w2.setHp(w2.getHp() + 50);
15   }
16 }
```

※このコードの前提

- Hero や Wizard、Thief は、抽象クラス Character を継承している。
- Character は name と hp フィールドおよびその getter／setter、attack() と run() メソッドを持つ。

このプログラムの宿泊処理（10 行目から 14 行目）には 2 つの課題があります。

【コードに重複が多い】

「〜 .setHp(〜 .getHp() + 50)」という同じ処理が何度も登場するため、記述がめんどうです。変数名の取り違えも発生するかもしれません。

【将来的に多くの修正が必要】

パーティの人数が増えた場合、宿泊処理に行を追加しなければなりません。また、インスタンス変数名が変更になった場合も、コードに修正が必要です。

しかし、多態性と配列を上手に組み合わせれば、この問題は解決します。次のリスト 13-7 を見てください。

リスト 13-7

Main.java

```
public class Main {
  public static void main(String[] args) {
    Character[] c = new Character[5];
    c[0] = new Hero();
    c[1] = new Hero();
    c[2] = new Thief();
    c[3] = new Wizard();
    c[4] = new Wizard();
    // 宿屋に泊まる
    for (Character ch : c) {              // 1名ずつ順に取り出し
      ch.setHp(ch.getHp() + 50);          // HP を 50 回復する
    }
  }
}
```

ポイントは3行目でCharacter配列を使っている点です。従来のように5人のインスタンスを厳密にHeroやThiefとして扱おうとする限り、それらを一括しては扱えません。しかし、それぞれをCharacterだとザックリ見なせば「どれもキャラクター」ですので、5つのインスタンスをCharacter配列にまとめ、ループを回して一括で処理することも可能になります。

図 13-12　さまざまな職業のキャラクターたちも、すべて Character 配列に格納して一括で処理できる

13.5.3 同一視してザックリとした引数を受け取る

多態性の活用法、もう１つ思いつきました！ ずっと気になっていたことが、これで一気に解決ですよ！

湊くんがずっと気にしていたのは、「勇者や魔法使いのattack()メソッドが、必ず次のような宣言になっていた」ということです（以下はMatangoの例です）。

```
public void attack(Matango m) {
  ：
}
```

「お化けキノコしか攻撃できないゲーム」なんてありえないから、今までHeroは次のようにしていたんですよ。

リスト 13-8

Hero.java

```
1  public class Hero extends Character {
2    public void attack(Matango m) {        ← お化けキノコ攻撃用
3      System.out.println(this.name + "の攻撃！");
4      System.out.println("敵に10ポイントのダメージをあたえた！");
5      m.hp -= 10;
6    }
7    public void attack(Goblin g) {         ← ゴブリン攻撃用
8      System.out.println(this.name + "の攻撃！");
9      System.out.println("敵に10ポイントのダメージをあたえた！");
10     g.hp -= 10;
11   }
12   // 以下スライム用など続く
13 }
```

この方法はうまくいきますが、コードの重複が多くメンテナンスが大変です。また、将来新たなモンスターが増えるたびにattack()メソッドも増やさなければなりません。そこで、attack()メソッドを次のように修正しましょう。

リスト 13-9

Hero.java

```
public class Hero extends Character {
  public void attack(Monster m) {      ← モンスター攻撃用
    System.out.println(this.name + "の攻撃！");
    System.out.println("敵に10ポイントのダメージをあたえた！");
    m.hp -= 10;
  }
}
```

2行目のattack()メソッドの引数に注目してください。「攻撃する相手は、**ザックリみれば何らかのモンスターであれば何でもいい**」という表明です。このようなattack()メソッドであれば、Monsterクラスを継承しているSlimeやGoblin、そして将来登場するモンスターたちも攻撃することができます。

図 13-13　異なるインスタンスを引数で同一視して受け取る

13.5.4 ザックリ利用しても、ちゃんと動く

なるほどなぁ。厳密には違う物を同一視することで、同じ配列に入れたり、同じ引数として処理したりできちゃうんだ。

そうだよ。でも「多態性の本当のすごさ」は、ただ単に処理をまとめられるという単純なものだけじゃないんだ。

多態性の真価は、これまで学んだ次の２つのことを組み合わせたときに現れます。

1. **ザックリ捉えてまとめて扱う**
   13.5.2 項（配列でまとめて扱う）や 13.5.3 項（引数でまとめて扱う）で紹介したように、厳密には異なるインスタンスをまとめて扱うことができます。

2. **メソッドの動作は中身の型に従う**
   13.3 節の最後に学んだように「インスタンスは何型の箱に入っていようと、自身の型のメソッドが動作する」という原則があります。

リスト 13-10

Main.java

```
public class Main {
  public static void main(String[] args) {
    Monster[] monsters = new Monster[3];
    monsters[0] = new Slime();
    monsters[1] = new Goblin();
    monsters[2] = new DeathBat();
    for (Monster m : monsters) {
      m.run();    // 同じ指示を繰り返す
    }
  }
}
```

実行結果

```
スライムは、体をうねらせて逃げ出した。
ゴブリンは、腕をふって逃げ出した。
地獄コウモリは、羽ばたいて逃げ出した。
```

13
章

ここまで学んだことを考えると、この動作は納得できます。
でもこれのどこが、そんなにスゴいんですか？

改めて全体を俯瞰することで、
多態性の構図と本質が見えてくるよ。

図 13-14　指示する側はいいかげん。動く側は自分のやり方で動く

コードの７～９行目で、指示を出す側（メソッドを呼び出す側）は、それぞれのモンスターに対して同じように「とにかく逃げろ」と、いいかげんな指示を繰り返しているだけです。

一方、モンスターたちは「逃げろ」と言われたら、きちんと**独自の方法で**逃げます。「どう逃げるか」については自分で理解していて、その方法（自分のクラスに定義された run() メソッドの内容）を使って逃げるのです。

このように、**呼び出し側は相手を同一視し、同じように呼び出す**のに、**呼び出される側は、きちんと自分に決められた動きをする**（同じ呼び出し方なのに、多数の異なる状態を生み出すことがある）という特性から「多態性」という名前が付けられています。

13.6 第13章のまとめ

この章では、次のようなことを学びました。

インスタンスをあいまいに捉える

- 継承により is-a の関係が成立しているなら、インスタンスを親クラス型の変数に代入することができる。
- 親クラス型の変数に代入することは、あいまいに捉えること。

「箱の型」と「中身の型」の役割

- どのメンバを利用できるかは、箱の型（対象をどう捉えているか）で決まる。
- メンバがどう動くかは、中身の型（対象が何であるか）で決まる。

捉え方の変更

- キャスト演算子を用いれば、厳密な型への強制代入ができる。
- 不正な代入が行われた場合、ClassCastException が発生する。

多態性

- 厳密には異なる複数のインスタンスを同一視して、親クラス型の配列にまとめて格納できる。
- 同様に、親クラス型の引数や戻り値を利用して、厳密には異なる対象をまとめて処理できる。
- 同一視して取り扱っても、個々のインスタンスは各クラスにおける定義に従い、異なる動作を行う。

13.7 練習問題

練習 13-1

次の図中の四角に入る適切なクラス名を考えてください。

	(1)	(2)
コード	Item i = new Sword();	[　　] a = new [　　] ();
イメージ図	[　　]インスタンス << [　　] >> i	[　　]インスタンス << Monsterです >> a
解説文	[　　] を生成したが、 ザックリと [　　] と と見なす。	Slimeを生成したが、 ザックリと [　　] と見なす。

練習 13-2

次のようにクラスが宣言されています。

A.java

```
1 public final class A extends Y {
2   public void a() { System.out.print("Aa"); }
3   public void b() { System.out.print("Ab"); }
4   public void c() { System.out.print("Ac"); }
5 }
```

B.java

```
1  public class B extends Y {
2    public void a() { System.out.print("Ba"); }
3    public void b() { System.out.print("Bb"); }
4    public void c() { System.out.print("Bc"); }
5  }
```

X.java

```
1  public interface X { void a(); }
```

Y.java

```
1  public abstract class Y implements X {
2    public abstract void a();
3    public abstract void b();
4  }
```

このとき、次の問いに答えてください。

①「X obj = new A();」として A インスタンスを生成した後、変数 obj に対して呼ぶことができるメソッドを、a()、b()、c() の中からすべて挙げてください。

②「Y y1 = new A(); Y y2 = new B();」として A と B のインスタンスを生成した後、「y1.a(); y2.a();」を実行した場合に画面に表示される内容を答えてください。

練習 13-3

練習 13-2 で用いた A クラスや B クラスのインスタンスをそれぞれ 1 つずつ生み出し、要素数 2 からなる単一の配列に格納するとします。格納した後には配列の中身をループで順に取り出し、それぞれのインスタンスの b() メソッドを呼ぶ必要があります。以上の前提に基づき、次の問いに答えてください。

①配列変数の型としては何を用いるべきか答えてください。
②問題文に記述された内容のプログラムを作成してください。

練習問題の解答

練習 13-1 の解答

（1）（左上から順に）Sword、Item、Sword、Item

（2）（左上から順に）Monster、Slime、Slime、Monster

練習 13-2 の解答

① a() メソッド

② AaBa

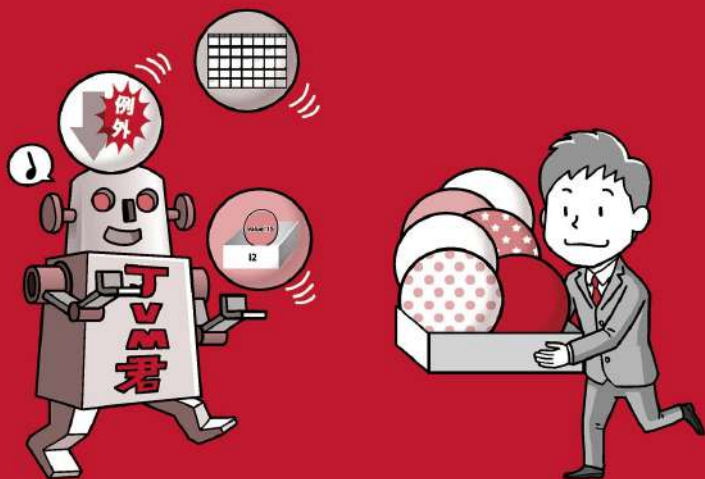
練習 13-3 の解答

① Y[] 型

②以下のリストを参照。

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Y[] array = new Y[2];
4      array[0] = new A();
5      array[1] = new B();
6      for (Y y : array) {
7        y.b();
8      }
9    }
10 }
```

第III部 もっと便利にAPI活用術

もっと楽しく、もっと便利に

読める…！　読めるぞっ…！！

どっ、どうしたの。急に PC の画面を指でなぞったりし始めちゃって…。

なるほど、第 I 部（p.247）で少しだけ紹介した Java の API リファレンスだね。

はい。あのときもこれを読もうとしたんですが、static とか extends とか出てきて全然意味がわかんなかったんです。でも今は、API を見ていたら意味や使い方が、ある程度ならば推測できるんです！

おめでとう。ふたりはもう、Java が用意した数万に及ぶ命令や高度な機能を自由に使える準備ができたんだ。すべては無理だけど、特に重要なものたちを紹介しておこうか。

ここまで学び終えた私たちは、Java の基本的な文法をほぼマスターしています。このことは、基本文法やオブジェクト指向を駆使して作成されている Java の API についても、私たちが理解可能であり活用できることを意味します。
最後の第 III 部では、多くの開発で必要となる API を中心に、Java プログラミングをより楽しく、より便利にしてくれる機能を紹介していきましょう。

第14章

Java を支える標準クラス

API に含まれるクラスや、そのメンバに関する解説は
API リファレンスに掲載されていますが、
初めて Java プログラムを学ぶ人にとって、
その解説は理解しやすいものではありません。
そこで本章以降では、Java が持つ多くの API の利用方法を
やさしく紹介していきます。

CONTENTS

14.1 日付を扱う

14.1.1 日時情報を扱う 2 つの基本形式

ここからは特に大事な API のクラスたちを紹介していくよ。まずは日付の情報を扱うための Date クラスから紹介しよう。

プログラムを開発していると、日付情報を扱う局面に多く遭遇します。たとえば ATM のプログラムであれば、「取引を行った時刻が、何年何月何日の何時何分何秒であるか」という情報を変数に入れて取り扱う必要があるでしょう。

Java でも日時情報を扱うことができますが、やや複雑な取り扱いが必要です。

そういえば RPG の開発で日付を使いたかったのですが、API リファレンスが難しくて理解できず、結局は諦めてしまいました。

Java では「日付情報」を表すために使う型（クラス）が 1 つではないということが重要なポイントだよ。

Java では日時の情報を表すための形式が 4 つあり、それぞれ用途に合わせて使い分ける必要があります。まずは基本となる 2 つの形式を学びましょう。

形式 1：long 型の数値

基準日時である 1970 年 1 月 1 日 0 時 0 分 0 秒（これを「エポック」といいます）から経過したミリ秒（1/1000 秒）数で日時情報を表現する方法です。たとえば、1316622225935 という long 値は「2011 年 9 月 22 日 1 時 23 分 45 秒」を意味します。

この long 型による形式はシンプルであるため、コンピュータにとってとても扱いやすく、JVM 内部のさまざまな部分で利用されています。たとえば、System.currentTimeMillis() メソッドを呼べば、現在日時を long 型で得られるため、リスト 14-1 のような「処理時間の計測」を簡単に行うことができます。

リスト 14-1

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      long start = System.currentTimeMillis();
4      // ここで何らかの時間がかかる処理
5      long end = System.currentTimeMillis();
6      System.out.println("処理にかかった時間は…"
          + (end - start) + "ミリ秒でした" );
7    }
8  }
```

しかし人間は、この long 値から「年・月・日・時・分・秒」を読み取ることができません。また、long 型は日付情報以外の数値の格納にも利用される型なので、必ずしも変数の中身が日時情報だと断定できないという問題もあります。

形式 2：Date 型のインスタンス

long 型の課題を克服するために広く用いられているのが java.util.Date クラスです。このクラスは、内部で long 値を保持しているだけなのですが、「Date 型の変数であれば、中身は日付情報である」と一目でわかるため、**Java で日時の情報を扱う場合に最も利用される形式**となっています。Date インスタンスを生成して利用するには、以下の構文を用います。

現在日時を持つ Date インスタンスの生成

Date d = new Date(); // 現在日時を持つインスタンス生成

指定時点の日時を持つ Date インスタンスの生成

Date d = new Date(long 値); // long 値の日時を持つインスタンス生成

Date インスタンスの内部に格納されている long 値を取り出したい場合は getTime() メソッドを、long 値をセットしたい場合は setTime() メソッドを用います。これら Date クラスの構文を用いて現在日時を表示するプログラムがリスト 14-2 です。

リスト 14-2

Main.java

```
import java.util.Date;
public class Main {
  public static void main(String[] args) {
    Date now = new Date();
    System.out.println(now);
    System.out.println(now.getTime());
    Date past = new Date(1316622225935L);
    System.out.println(past);
  }
}
```

（1行目）import しておくと便利
（4行目）現在の日時を取得

実行結果

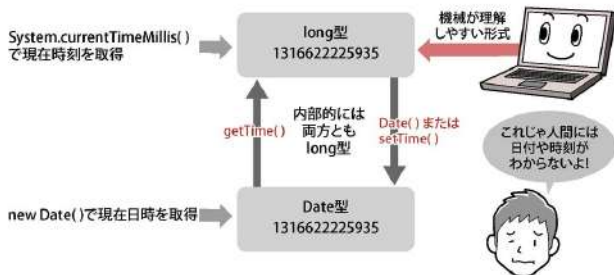
```
Fri Aug 12 16:05:55 GMT+09:00 2011
1313132755277
Thu Sep 22 01:23:45 GMT+09:00 2011
```

（※実行の日時により以上の日付と数値は変わります）

ここで、これまでに学習した 2 つの形式について図 14-1 に整理します。

java.sql.Date と混同しない

java.util.Date と似たクラスとして、java.sql.Date というクラスがあります。これはデータベースを利用したプログラムで利用するもので、通常の日時情報の格納には使いません。

図 14-1　long 値と Date 型の関係

14.1.2 人間が扱いやすい 2 つの形式

long 値も Date クラスも、結局はエポックからの経過ミリ秒数を扱っていることに違いはなく、人間が日付情報を扱うには不便です。そこで、人間にとって使いやすい 2 つの日時形式を紹介します。

形式 3：人間が読みやすい String 型のインスタンス

人間が読みやすいのは「2011 年 9 月 22 日 1 時 23 分 45 秒」のような文字列としての形式です。画面に時刻を表示する場合、この形式に変換する必要があります。

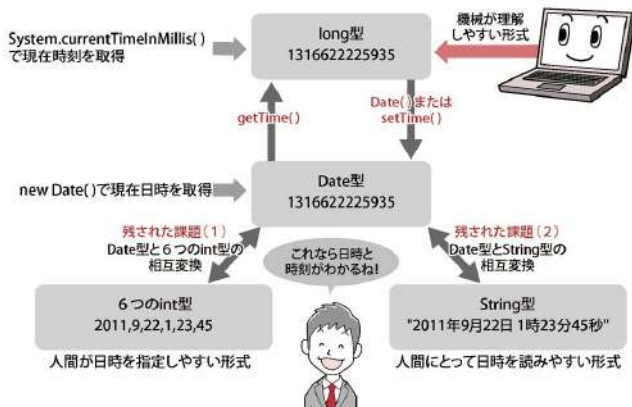
ただし、一口に文字列といっても、さまざまな形式が考えられる点には考慮が必要です。たとえば「2011/9/22 1:23:45」という形式で表示したいこともあるでしょうし、あるいは「11-09-22 01:23:45AM」と表現したいこともあるでしょう。

形式 4：人間が指定しやすい「6 つの int」形式

前述の System.currentTimeMiilis() メソッドを使うか、Date クラスを new すれば「現在の時刻」は簡単に得られます。しかし、ある特定の日時情報（たとえば 2011 年 9 月 22 日 1 時 23 分 45 秒）を人間がキーボードやマウスなどで入力する場合には、「年・月・日・時・分・秒」をそれぞれ整数（int 値）として指定することが一般的です。

ここまでで、私たちは日時を表す 4 つの形式を学びました。機械が扱いやす

い形式としての long 型と Date 型、人間が扱いやすい形式としての文字列型と 6 つの int 値です（図 14-2）。

図 14-2　long 値と Date 型の関係

これら 4 つの形式を自由に利用できるようになるために、「(1) Date 型と 6 つの int 値の相互変換」と「(2) Date 型と文字列の相互変換」の 2 つを学びましょう。

14.1.3　Calendar クラスの利用

「課題 (1) Date 型と 6 つの int 値の相互変換」の解決のためには、java.util.Calendar クラスが準備されており、構文は次のとおりです。また、そのサンプルをリスト 14-3 に示します。

「6 つの int 値」から Date インスタンスを生成する

```
Calendar c = Calendar.getInstance();
c.set( 年 , 月 , 日 , 時 , 分 , 秒 ); または c.set(Calendar.〜 , 値 );
Date d = c.getTime();
```

※〜には YEAR、MONTH、DAY_OF_MONTH、HOUR、MINUTE、DAY などを指定する。

Date インスタンスから「6 つの int 値」を生成する

```
Calendar c = Calendar.getInstance();
c.setTime(d);   // d は Date 型変数
int year = c.get(Calendar.YEAR);
int month = c.get(Calendar.MONTH);
int day = c.get(Calendar.DAY_OF_MONTH);
int hour = c.get(Calendar.HOUR);
int minute = c.get(Calendar.MINUTE);
int second = c.get(Calendar.SECOND);
```

リスト 14-3

Main.java

```
1  import java.util.Calendar;
2  import java.util.Date;
3  public class Main {
4    public static void main(String[] args) {
5      // 現在の年を表示する
6      Date now = new Date();
7      Calendar c = Calendar.getInstance();
8      c.setTime(now);
9      int y = c.get(Calendar.YEAR);
10     System.out.println("今年は" + y + "年です");
11     // 指定した日のDate型の値を得る
12     c.set(2010,8,22,1,23,45);
13     c.set(Calendar.YEAR, 2011);    ← 年だけを 2011 に変更
14     Date past = c.getTime();
15   }
16 }
```

「月」の値にご用心

Calendar を用いて「月」の情報を取得・設定する場合には、1～12 ではなく 0～11 で指定することになっているため注意が必要です。たとえば、2 月を設定したい場合には、「c.set (Calendar.MONTH, 1)」とします。

14.1.4 SimpleDateFormat クラスの利用

最後に残された「課題 (2) Date 型と文字列型の相互変換」の解決には、簡単な方法と高度な方法があります。「Sun Aug 07 16:38:59 JST 2011」のような文字列（多少、読みづらいですが）でよければ、Date インスタンスの toString() メソッドを呼び出すだけで取得できます。

もし、独自に書式を指定して「2011 年 8 月 7 日 16 時」のような文字列を生成したい場合は、java.text.SimpleDateFormat クラスを用いる必要があります。

Date から String を生成する

```
SimpleDateFormat f = new SimpleDateFormat ( 書式文字列 );
String s = f.format (d);    // d は Date 型変数
```

String から Date を生成する

```
SimpleDateFormat f = new SimpleDateFormat ( 書式文字列 );
Date d = f.parse ( 文字列 );
```

この「書式文字列」には何を指定すればいいんですか？

"yyyy/MM/dd" や "yyyy 年 MM 月 dd 日 " のように、日付の書式を指定するんだよ。

書式文字列に使うことができる主な記号は以下のとおりです。

表 14-1　書式文字列として利用可能な文字（一部）。

文字	意味
y	年
M	月
d	日
E	曜日
a/p	午前／午後
H	時（0～23）
K	時（0～11）
m	分
s	秒

次のリスト 14-4 は、SimpleDateFormat を用いて指定した形式で日付情報を表示するコード例です。

リスト 14-4

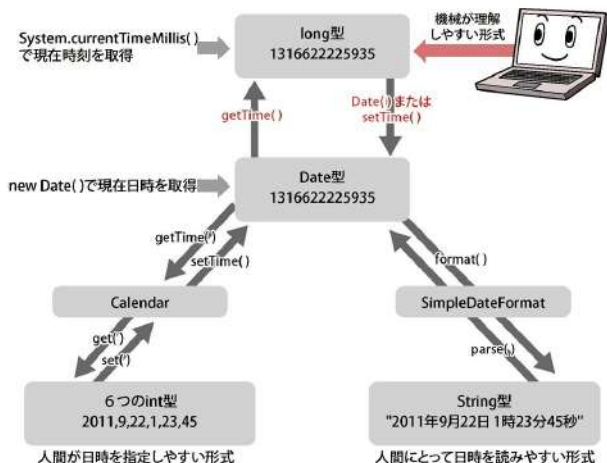
Main.java

```
1  import java.text.SimpleDateFormat;
2  import java.util.Date;
3  public class Main {
4    public static void main(String[] args) throws Exception {
5      // 本日の日時を表示する
6      Date now = new Date();
7      SimpleDateFormat f =
```

throws については 15 章で解説します

```
            new SimpleDateFormat("yyyy/MM/dd HH:mm:ss" );
 8      String s = f.format(now);
 9      System.out.println(s);
10      // 指定日時の文字列を解析しDate型として得る
11      Date d = f.parse("2011/09/22 01:23:45");
12    }
13  }
```

図 14-3 は 4 つの形式をまとめたものだ。日時情報は Date 型で扱う方法を基本とし、必要に応じてほかの形式に変換して使ってほしい。

図 14-3　long 値と Date 型の関係

ここまで紹介してきたように、Java における日付の取り扱いは少し複雑です。また、あいまいな日時情報や時差情報をうまく扱うこともできません。

Java8 以降では、上記のような課題を克服した新しい日付 API が追加されています。

14.2 すべてのクラスの祖先

14.2.1 暗黙の継承

14.1.4 節では Date インスタンスには toString() というメソッドがあることに触れたね。実は、この toString() というメソッドは、**すべてのクラス**で利用できるんだ。

すべてって、Hero や Matango にもですか？ でもボクたち、Hero クラスには、そんなメソッドを宣言していませんよ？

湊くんが不思議に思うのもしかたありません。しかし、次のリスト 14-5 は何の問題もなくコンパイルでき、実行が可能です。

リスト 14-5

Empty.java

```
1 public class Empty {}
```

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Empty e = new Empty();
4     String s = e.toString();
5     System.out.println(s);
6   }
7 }
```

メソッドもフィールドもいっさい定義していないクラスの toString() を呼び出せるのは、Java には次のような約束が備わっているからです。

暗黙の継承

あるクラスを定義するとき、extends で親クラスを指定しなければ、java.lang.Object を親クラスとして継承したと見なされる。

つまり、先ほどの Empty クラスは、以下のようなクラス定義と実質的に同じものなのです。

Empty.java
```
public class Empty extends Object {}
```

extends を指定しなくても必ず Object を継承するということは、「Java では親なしのクラスを定義できない」ということにほかなりません。これまで紹介してきたさまざまなクラスも、その親クラスを順に辿っていくと、最終的には java.lang.Object クラスに到達します。

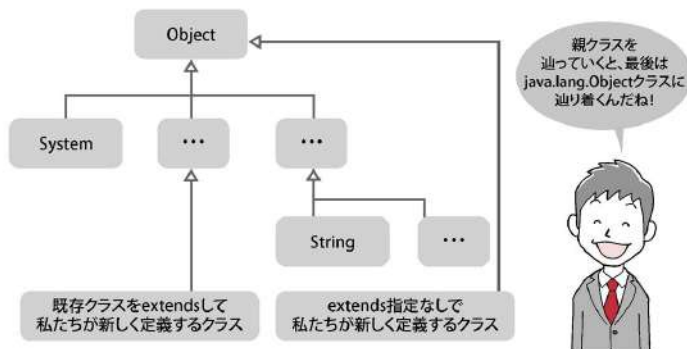



図 14-4　java.lang.Object はすべてのクラスの祖先

API リファレンスを調べると、全クラスの祖先である Object クラスには、次のようなメソッドが定義してあることがわかります。

・equals()　あるインスタンスと自分自身とが同じかを調べる
・toString()　自分自身の内容の文字列表現を返す

Empty クラスで toString() を利用できたのは、「暗黙の親クラスである Object クラスから継承していたから」だったのです。

14.2.2 Object クラスの存在価値

Java では、「すべてのクラスは絶対に java.lang.Object の子孫」なんですね。

でも、そもそも Java を作った人は、なぜ「すべての先祖 Object クラス」なんていうものを作ったんだろう。

いい疑問だね。それでは Object クラスが必要な理由や、その存在価値を紹介しよう。

なぜ Java には、java.lang.Object のような「全クラスの祖先」にあたるクラスが準備してあるのでしょうか。その理由としては次の 2 つが考えられます。

理由 1：多態性を利用できるようになるから

すべてのクラスが Object を先祖に持つのですから、「すべてのクラス is-a Object」ということができます。あらゆるクラスは「ザックリみれば、どれも Object」として同一視できるのです（第 13 章を参照）。

これは、次のリスト 14-6 のように、「Object 型の変数には、どんなインスタンスも代入できる」ことを意味しています。

リスト 14-6

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
```

```
3      Object o1 = new Empty();
4      Object o2 = new Hero( );
5      Object o3 = "こんにちは";
6    }
7  }
```

また次のリスト 14-7 のように、引数として Object 型を用いることで、「何型でもいいからインスタンスを渡せる」メソッドを作ることもできます。

リスト 14-7

Main.java

```
1  public class Main {
2    public void printAnything(Object o) {
3      // 何型でもいいから、引数を1つ受け取り画面に表示
4      System.out.println(o.toString());
5    }
6  }
```

System.out.println() の中身

ちなみに、リスト 14-7 の printAnything() メソッドとほぼ同じ内容を持つのが、私たちがいつも利用している System.out.println() メソッドです。API リファレンスで System.out.println() を調べると、引数として Object型を受け取ることができるようになっていることがわかります。

System.out.println() は渡されたインスタンスの内容を画面に表示する役割を持っていますが、その実現のために、渡されたオブジェクトの toString() メソッドを呼んで文字列表現を得て、それを画面に出力しています。

このようにObject型の変数は、あらゆる参照型のインスタンスを格納できます。格納できないのは、基本データ型（intやlongなど）の情報だけです。

理由2：すべてのクラスが最低限備えるべきメソッドを定義できるから

「少なくともJavaのクラスであれば、最低限備えておいたほうがよい機能」というものがあります。たとえば、インスタンス同士の内容が同じものかをチェックしたり、インスタンスの内容がどのようなものかを文字情報として表示させたりしたいことは頻繁にあります。

Objectクラスでequals()やtoString()などが定められているおかげで、私たちはクラスの種類を気にすることなく、常に同じ方法で内容を比較したり表示したりできるのです。

14.2.3 デフォルトの文字列表現

さて突然ですが、次のコードを見て表示される結果を想像してみてください。

リスト 14-8

Hero.java

```
1 public class Hero {
2   String name;
3   int hp;
4     ⋮
5 }
```

Main.java

```
1 public class Main {
2   public static void main(String[] args) {
3     Hero h = new Hero();
4     h.name = "ミナト";
5     h.hp = 100;
6     System.out.println(h.toString());
7   }
8 }
```

引数は単にhでもよい

14章

toString() メソッドは、オブジェクトの中身の情報を文字列にしてくれるのよね。「名前 = ミナト、HP=100」みたいな表示が出るのかな。

ボクは「名前：ミナト／ HP：100」だと思うな。

2 人の予想とは大きく異なり、実行結果は次のようなものになります。

実行結果

```
Hero@3487a5cc
```

※@以降は実行環境ごとに異なります。

あれ？ なんでこんな変な出力になっちゃうんだろう。

試しに Date クラスを new して toString() を呼んでみましたけど、こんなおかしな表示ではなく、ちゃんと日時が表示されました。どうして Hero クラスだけ？

リスト 14-8 でわかるように、Hero クラスには toString() メソッドが宣言されていません。ということは、Main.java の 6 行目で呼び出されて動作しているのは、Object クラスに宣言され、Hero クラスに継承されてきた toString() メソッドです。

実際、Object クラスに定義されている toString() メソッドは、「型名 @ 英数字」という形式で情報を表示するという極めてシンプルな処理内容になっています。

14.2.4 文字列表現を定義する

「println(h.toString())」だけで「名前：ミナト／ HP：100」みたいな表示をしてほしいです。

そのためには「toString() が呼ばれたら、どのフィールドの内容を、どう修飾して文字列表現にするか」を湊くんが指示してあげる必要があるね。

Hero クラスの toString() メソッドが呼ばれた際、湊くんが期待したような文字列を得るには、次のように Hero クラスで **toString() メソッドをオーバーライド**する必要があります。

リスト 14-9

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4      ⋮
5    public String toString() {        ← オーバーライドする
6      return "名前：" + this.name + "／HP：" + this.hp;
7    }
8  }
```

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h = new Hero();
4      h.name = "ミナト";
5      h.hp = 100;
6      System.out.println(h);        ← この 1 行で内容を表示
7    }
```

```
8 }
```

Date クラスの toString() を呼び出したときに変な表示にならなかったのは、きっと Date クラスでオーバーライドされていたからなのね。

このように toString() メソッドをオーバーライドしておくことで、インスタンスの内容を画面に出力することが簡単にできるようになります。いくつもの情報を内部に持つようなクラスを開発したら、ぜひ toString() メソッドをオーバーライドすることを検討してください。

14.2.5 等値と等価の違い

Object クラスで定義されているメソッドの中でも、toString() と並んで有名なのが equals() メソッドです。equals() メソッドは、2 つのインスタンスが「同じ内容であるか」を判定するため、次のように用いられます。

リスト 14-10

Main.java

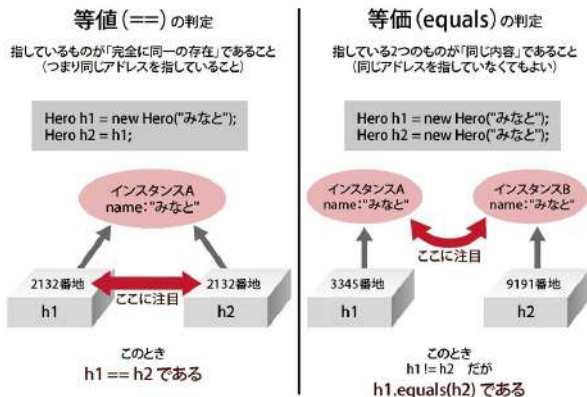
```
1  public class Main {
2    public static void main(String[] args) {
3      Hero h1 = new Hero();        ← 1人目の勇者
4      h1.name = "ミナト";
5      h1.hp = 100;
6      Hero h2 = new Hero();        ← 2人目の勇者
7      h2.name = "ミナト";
8      h2.hp = 100;
9      if (h1.equals(h2) == true) {
10       System.out.println("同じ内容です");
11     } else {
12       System.out.println("違う内容です");
```

```
13          }
14      }
15  }
```

同じかどうかの判断って「if (h1 == h2)」ではダメなんですか？

そういえば第 3 章で「文字列の比較は equals() を使う」と丸暗記しましたけど、"equals" と "==" って何が違うんでしょう？

湊くんがいうように、同じかどうかの判定には、「if (h1==h2)」という書き方もあります。しかし、**== を使った判定と equals を使った判定では、微妙に意味が異なる**ことに注意が必要です。前者のような比較は**等値**(equality)であるか、後者は**等価**(equivalence)であるかを判定するためのものです。

図 14-5　等値と等価の違い

だから String 型は equals() で判定が必要だったのね。

14.2.6 等価判定方法の指定

equals() は、すべてのクラスで使えるので、等価か調べたいときには、とにかく equals() を呼び出せばいいんですね！

いや、equals() についても湊くんが「何をもって同じものと見なすか」を指定してあげないと正しく動かないんだ。

何の準備もなく自分で作ったクラスの equals() メソッドを呼び出しても、うまく動きません。たとえば先ほどのリスト 14-10 では内容が同じはずの 2 つの Hero インスタンスを比較していますが、実行すると画面には「違う内容です」と表示されてしまいます。

実は Object クラスから継承される equals() メソッドの処理内容は、おおむね以下のようなものになっています。

```
public boolean equals(Object o) {
  if (this == o) { return true; }
  else { return false; }
  }
```

これって…ただの等値判定じゃないですか！

そうなんだ。「何をもって同じ内容と見なすか」は、それぞれのクラスによって異なるから、Object クラスでは「とりあえず等値なら true を返す」作りにしてあるんだよ。

そもそも等価の判定は、機械的には行うことができないものです。なぜなら**「何をもって、意味的に同じものと見なすか」は、クラスによって異なり、一律には決められない**からです。たとえば、名前が「ミナト」の勇者と「ミナト」の勇者を同じものと見なすかどうかは、作るゲームによって異なるでしょう。

そこでクラスの開発者は、そのクラスのインスタンスについて、「何をもって、意味的に同じと見なすか」をequals()メソッドのオーバーライドという形で指定しなければなりません。

仮に、Heroクラスは「名前が同じであれば同じ内容のインスタンスと見なす」と定義するならば、次のようなコードになるでしょう。

リスト 14-11

Hero.java

```
1  public class Hero {
2    String name;
3    int hp;
4     ⋮
5    public boolean equals(Object o) {
6      if (this == o) { return true; }
7      if (o instanceof Hero) {
8        Hero h = (Hero) o;
9        if (this.name.equals(h.name)) {
10         return true;
11       }
12     }
13     return false;
14   }
15 }
```

等値なら間違いなく等価

名前が等しければ等価

Heroクラスに上記のような修正を行った上でリスト14-10のMainクラスを実行すると、「同じ内容です」という表示結果を得られます。

toString()とequals()のオーバーライド

新しくクラスを開発したら、toString()とequals()をオーバーライドする必要性がないかを検討する。

14.3 基本データ型をオブジェクトとして扱う

14.3.1 ラッパークラスとは

先輩。API リファレンスを見ていたら、java.lang パッケージの中に、Long とか Boolean とか、おもしろい名前のクラスを見つけました。

それらは「ラッパークラス」と呼ばれるクラスだよ。

これまでに学んだように、Java で利用できる型は、大きく「基本データ型」と「参照型」の 2 種類に分けられます。特に int 型や boolean 型といった基本データ型については、この本の冒頭から数多く使ってきました。

ところで、Java の API では、それぞれの基本データ型に対応したクラスを準備しており、それらは**ラッパークラス**（wrapper class）と総称されています。

表 14-2　基本データ型とラッパークラスの対応

基本データ型	ラッパークラス
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

これらラッパークラスには2つの役割があります。

第1の役割：便利なメソッドの提供

最初の役割は、「ある基本データ型に関する、便利なメソッドを提供すること」です。たとえば、Integerクラスは以下のようなintにまつわる多くの便利なメソッドを持っています。

表14-3　Integerクラスが持つ便利なメソッド（一部）

メソッド名	ふるまい
parseInt()	数字の文字列をint型に変換する
toHexString()	整数を16進数表現に変換する

これらの便利メソッドは、いずれも静的メソッド(static)で宣言されているため、ラッパークラスをインスタンス化することなく、手軽に利用できます。

第2の役割：インスタンスとして扱えるようにする

第2の役割は、基本データ型の情報をインスタンスとして扱えるようにすることです。次のコードを見てください。

```
int i1 = 15;
Integer i2 = Integer.valueOf(i1);   int → Integer変換
System.out.println(i2);
```

このコードの2行目では、IntegerクラスのvalueOfメソッドを呼び出し、**基本データ型intの変数i1の内容を、ラッパークラス型Integerの変数i2に変換**しています。

この変数i2に格納されているIntegerインスタンスは、内部で「15」という値を保持するだけの極めてシンプルなインスタンスです。HPや名前など、多数の情報を保持していたHeroインスタンスなどとは大違いですね。

図 14-6　基本データ型とラッパークラス

図 14-6 のどちらの変数も、結局「15」という情報を持っています。箱の中に入っているものが数値そのものか、インスタンスに包まれた数値なのかの違いでしかありません。

しかも、基本データ型のほうが、よりメモリの消費量が少なく、高速に読み書きできることを考えると、Integer 型のようなラッパークラスを使う必要性を感じられないかもしれません。

しかし、Java には**「基本データ型を利用できない API」がいくつか存在します。**そのような API のメソッドを私たちが利用する場合、「15」という数字情報を渡したいときには、int 型の変数ではなく Integer 型を利用する必要があるのです。

14.3.2 基本データ型とラッパークラスの相互変換

第 2 章で学んだように、基本的に異なる型の変数に値は代入できません。よって「15」という数字を int 型の変数に代入することはできても、Integer 型の変数にそのまま代入することはできません。

つまり「int 型の変数に入っている値を Integer 型の変数に入れたい場合」や、逆に「Integer 型の変数に入っている値を int 型の変数に入れたい場合」は、相互に型を変換する必要があります。

ラッパークラスへの変換には valueOf を、基本データ型への変換には intValue() や longValue() などの「～ Value()」メソッドを利用します。

リスト 14-12

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      int i1 = 15;
4      Integer i2 = Integer.valueOf(i1);
5      int i3 = i2.intValue();
6    }
7  }
```

int → Integer 変換

Integer → int 変換

「valueOf() や intValue() を呼ぶのは理解できたけど、ちょっとめんどうですよぉ～」とか、湊は思っているんじゃない？

そ、そんなことは断じて…ある…かも。

湊くんの感覚は正しいよ。だから Java には相互変換を自動的に行うしくみが備わっているんだ。

古いバージョンの Java では、「基本データ型とラッパークラスとの間の相互型変換」を valueOf() と～value() メソッドを用いて明示的に行う必要がありました。しかし Java5 以降では、暗黙的に相互変換を行う **AutoBoxing** と **AutoUnboxing** というしくみが導入されました。

AutoBoxing と AutoUnboxing

ラッパークラス型と基本データ型との間で代入を行う式を記述すると、自動的に valueOf() や～ value() による型変換が行われる。

このしくみのおかげで、次のリスト 14-13 を記述しても文法エラーが発生することなくコンパイルできます。ぜひ、リスト 14-12 と見比べてください。

リスト 14-13

Main.java

```
public class Main {
  public static void main(String[] args) {
    int i1 = 15;
    Integer i2 = i1;    // int → Integer 自動変換
    int i3 = i2;        // Integer → int 自動変換
  }
}
```

どうせ似たものだからと「空気を読んで」変換してくれるのね。

これでラッパークラスも使えるようになったから、いつ「基本データ型では使えない一部の API」を使う日がきても大丈夫だ！

そうだね。きっとその日も遠いことではないよ。

14.4 第 14 章のまとめ

この章では、次のようなことを学びました。

日付の扱い

- Java における日付情報は基本的に java.util.Date 型で扱う。
- その他、必要に応じて long 値、6 つの int、String 型に変換して用いる。
- 「年月日時分秒」の 6 つの int 値から Date インスタンスを得るためには Calendar クラスを使う。
- Date インスタンスの内容を任意の書式で文字列に整形したい場合は、SimpleDateFormat クラスを使う。

Object クラス

- Java において、すべてのクラスは Object クラスの子孫である。
- すべてのインスタンスは Object 型変数に格納可能である。
- すべてのクラスは Object から toString() や equals() を継承している。
- 自分で作成したクラスにおいては、文字列表現や等価判定方法を指定するため、toString() や equals() をオーバーライドする。

ラッパークラス

- 基本データ型に対応したラッパークラスが java.lang パッケージに存在する。
- 基本データ型とラッパークラスのデータは、valueOf() や ～ value() メソッドで明示的に変換できる。
- 両者は AutoBoxing ／ AutoUnboxing 機構により暗黙的にも変換される。

練習問題

練習 14-1

main() メソッドのみを持つクラス Main を定義し、以下の手順を参考にして「現在の 100 日後の日付」を「西暦 2011 年 09 月 24 日」という形式で表示するプログラムを作成してください。

① 現在の日時を Date 型で取得します。
② 取得した日時情報を Calendar にセットします。
③ Calendar から「日」の数値を取得します。
④ 取得した値に 100 を足した値を Calendar の「日」にセットします。
⑤ Calendar の日付情報を Date 型に変換します。
⑥ SimpleDateFormat を用いて、Date インスタンスの内容を表示します。

練習 14-2

口座番号を表す String 型フィールド accountNumber と、残高を表す int 型フィールド balance を持つ銀行口座クラスを作ってください。さらに、このクラスにメソッド宣言を追加し、次の①と②の条件を満たすように修正してください。

①口座番号 4649、残高 1592 円の Account インスタンスを変数 a に生成し、「System.out.println (a);」を実行すると、画面に「¥1592（口座番号＝ 4649）」と表示されること。
② 口座番号が等しければ等価と判断されること。ただし、「 4649」など、口座番号の先頭に半角スペースが付けられたものは、それを無視して比較すること（「 4649」口座と「4649」口座は同じものと捉える）。
（ヒント：java.lang.String クラスの trim () メソッドを利用します。）

14.6 練習問題の解答

練習 14-1 の解答

Main.java

```
1  import java.text.SimpleDateFormat;
2  import java.util.Calendar;
3  import java.util.Date;
5  public class Main {
6    public static void main(String[] args) {
7      // ①現在の日時をDate型で取得
8      Date now = new Date();
9      Calendar c = Calendar.getInstance();
10     // ②取得した日時情報をCalendarにセット
11     c.setTime(now);
12     // ③Calendarから「日」の情報を取得
13     int day = c.get(Calendar.DAY_OF_MONTH);
14     // ④取得した値に100を足してCalendarの「日」にセット
15     day += 100;
16     c.set(Calendar.DAY_OF_MONTH, day);
17     // ⑤Calendarの日付情報をDate型に変換
18     Date future = c.getTime();
19     // ⑥指定された形式で表示
20     SimpleDateFormat f =
           new SimpleDateFormat("西暦yyyy年MM月dd日");
21     System.out.println(f.format(future));
22   }
23 }
```

練習 14-2 の解答

Account.java

```
1  public class Account {
2    String accountNumber;     // 口座番号
3    int balance;              // 残額
4    /* ①文字列表現のメソッド */
5    public String toString() {
6      return "¥¥" + this.balance +
           "（口座番号：" + this.accountNumber + "）";
7    }
8    /* ②等価判定のメソッド */
9    public boolean equals(Object o) {
10     if (this == o) {
11       return true;
12     }
13     if (o instanceof Account) {
14       Account a = (Account) o;
15       String an1 = this.accountNumber.trim();
16       String an2 = a.accountNumber.trim();
17       if (an1.equals(an2)) {
18         return true;
19       }
20     }
21     return false;
22   }
23 }
```

第15章

例外

プログラムを設計するにあたっては、
実行時に想定外の事態が発生する可能性があることを
考慮に入れておく必要があります。
そして、そのような場合にも異常終了や誤動作しないよう
備えておかなければなりません。
Java には想定外の事態に対処する「例外」が備わっています。
本章をしっかり学習し、どのような状況でも安定して動く
高品質のプログラムを開発できるようになりましょう。

CONTENTS

15.1 エラーの種類と対応策

15.1.1 不具合のないプログラムを目指す

最近やっと Java に慣れてきて、エラーがあまり出ないようになりました！

それはよかった。でも重要なプログラムを作る場合は「あまり」ではダメだよ。

Java が誕生して 20 年以上が経ち、今では金融機関や官公庁などの社会基盤を支える情報システムを作る際にも Java が使われるようになりました。しかし、これらの大規模なシステムに不具合が生じると、時に大きな社会問題や損害賠償につながる可能性があります。

ある程度の経験を積めば「とりあえず動作するプログラムを作ること」は難しくありません。本当に難しいのは、「想定外の事態やユーザーの誤操作などがあっても、エラーを起こさず正常に動作するプログラムを作ること」なのです。

不具合のないプログラムを目指す

動くコードは書けて当たり前。不具合対策こそが、腕の見せどころ。

キビシイなぁ…。

何、言ってるのよ。途中で止まっちゃうゲームなんて、誰も遊んでくれないわよ！

15.1.2 3種の不具合と対処法

プログラムが想定どおりに動かないことを総じて不具合と言いますが、Javaプログラムの場合は大きく3つに分類できます。

①文法エラー（syntax error）

文法の誤りによりコンパイルに失敗します。代表例はセミコロン忘れ、変数名の間違い、privateメソッドを外部から呼び出す、などです（図15-1）。

```
>javac SyntaxError.java

SyntaxError.java:4: ';' がありません。

      }
      ^

エラー 1 個
```

最も単純なエラーだね。
ボクもよく間違えるよ

図15-1　文法エラーの例

②実行時エラー（runtime error）

実行している最中に何らかの異常事態が発生し、動作が継続できなくなるエラーです。Javaの文法としては問題がないためコンパイルは成功し、実行もできますが、実行中にエラーメッセージが表示されて強制終了します。代表例は配列の範囲外要素へのアクセス、0での割り算、存在しないファイルのオープン、などです（図15-2）。

```
>javac RuntimeError.java
>java RuntimeError
プログラムを開始します。処理を3つ実行します。
処理1を完了。
処理2を完了。

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 17
at RuntimeError.main(RuntimeError.java:11)
```

実行時例外は
コンパイルのときには
わからないから少しめんどうよね

図15-2　実行時エラーの例

③論理エラー（logic error）

Javaの文法に問題はなく、強制終了もしません。しかし、プログラムの実行結果が想定していた内容と違っています。代表例は、電卓ソフトを作ったが計算結果がおかしい、などです（図15-3）。

```
>javac LogicalError.java
>java LogicalError
プログラムを開始します。
3+5を計算します。
計算完了： 答えは35
プログラムを正常終了します。
```

論理的な誤りがある
一番やっかいな不具合だね

図15-3　論理エラーの例

開発者は、これら3種類の不具合に対して、それぞれ異なる対策を行う必要があります。その不具合の検出と解決についてまとめたものが次の図15-4です。

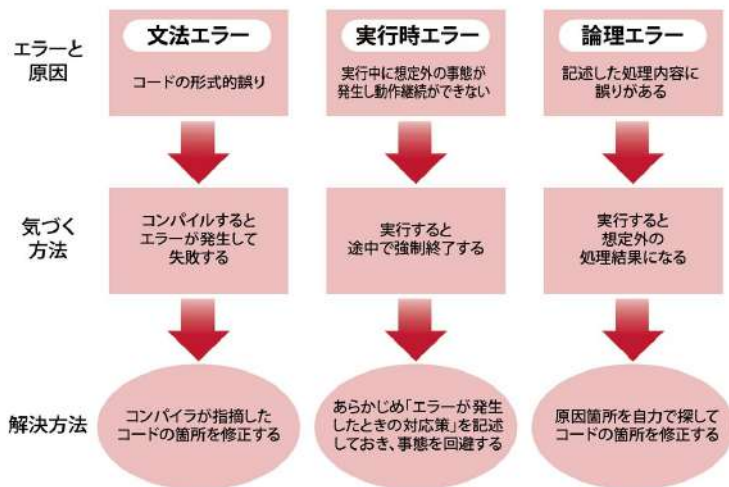



図15-4　各不具合に対する解決方法

文法エラーと論理エラーは、対策が似ていますね。

いいところに気づいたね。では、残る実行時エラーについて、少し深く考えてみよう。

15.1.3 例外的状況

文法エラーと論理エラーは「開発者の過失」であって、**開発者が開発時にテストをしっかり行い、コードを修正することで、本番での発生を予防できるもの**です。しかし、実行時エラーはそうはいきません。

そもそも実行時エラーは、「プログラム実行中に想定外の事態が発生したこと」によって起こります。この「想定外の事態」のことを**例外的状況**（exceptional situation）または単に**例外**（exception）といいます。例外的状況には次のようなものがあります。

■パソコンのメモリが足りなくなった

開発用のコンピュータには十分な容量のメモリがあったが、本番用コンピュータのメモリが少なく、動作中にメモリが足りなくなってしまった。

■存在すべきファイルが見つからない

動作中にdata.txtというファイルを読み込んで動くプログラムを開発したが、利用者が誤ってファイルを削除してしまった。

■ nullが入っている変数を利用しようとした

ユーザーが想定外の操作を行ったことが原因で、本来は変数に入るはずのない値（nullなど）が入り、その変数を使用するメソッドを呼び出してしまった。

これらすべての状況に共通するのは、**プログラマがソースコードを作成する時点では例外的状況の発生を予防できない**ということです。

「null が入っている変数を利用する」という例外的状況は、事前に if 文でチェックすれば「予防できる」と思いますけど？

朝香さんはプログラムに含まれる、すべてのメソッド呼び出しで null チェックを行うのかい？

あ…理論的には可能でも現実的ではないですね。

プログラマが「例外的状況の発生を防ぐこと」は困難です。しかし、プログラマは無力ではありません。「もし例外が発生したときに、どのように対処するか」という対策を用意しておくことが可能です。たとえば、「もし目的のファイルが見つからなければ、ユーザーに代わりのファイル名を入力してもらう」など、異常事態に備えた代替策を準備しておけばよいのです。このように例外的状況に備えて対策を準備し、その状況に陥った際に対策を実施することを**例外処理**（exception handling）と呼びます。

15.2 例外処理の流れ

15.2.1 従来型例外処理の問題点

パソコンでプログラムを動かしていたら強制終了したことがありました。きっと作者は例外処理をしていなかったのね。

そうかもしれないね。でも、昔は例外処理をきちんと書くことが、とても大変だったんだ。

Java が生まれる以前から例外処理はプログラマにとって取り組むべき重要な課題でした。たとえば Java の祖先にあたる C 言語の場合、「ファイルに『hello!』と書き込む」だけの簡単なプログラムは図 15-5 のように書きます。

```
/* ファイルを開く(失敗したら戻り値は定数NULL) */
FILE fp = fopen("c:¥¥test.txt");          本来の処理
if(fp == NULL) {
  printf("エラーです。終了します。");        例外処理
  exit(1);
}

/* ファイルに文字列を書き込む*/
fputs("hello!", fp);                      本来の処理

/* ファイルを閉じる(失敗したら戻り値は定数EOF) */
int c = fclose(fp);                       本来の処理
if(fp == EOF) {
  printf("エラーです。終了します。");        例外処理
  exit(1);
}
```

図 15-5　C 言語での例外処理の例

実はこのプログラムの「本来の処理」は、①ファイルを開く、②ファイルに文字を書き込む、③ファイルを閉じる、この 3 つだけです。しかし、命令を呼び出すたびに「もしもファイルが開けなかったら…」あるいは「もしもファイルに書けなかったら…」などの例外的状況を 1 つひとつチェックしているため、プログラムがわかりにくくなっています。そのため、C 言語のような古いプログラミング言語の例外処理には次のような問題点があります。

- **「本来の処理」が、どの行なのかわかりづらい。**
- **命令を呼び出すたびに 1 つひとつチェックしなければならない。**
- **めんどうなので例外処理をサボって書かないおそれがある。**

本来たった 3 行のプログラムがこんなに長くなってしまうなんて…。本格的なプログラムならもっと酷いことになりそう…。それで、つい例外処理をサボったプログラムになってしまうのね。

このような問題が発生するのは、従来型のプログラミング言語が**例外的状況発生の検知と対応に関する全責任をプログラマに求めている**からです。

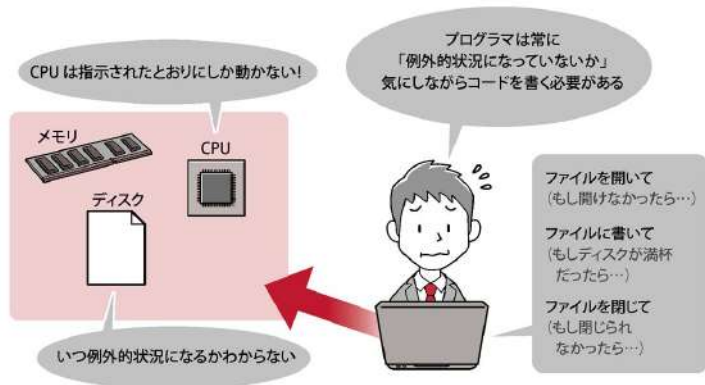



図 15-6　従来型のプログラミング言語での例外処理ではプログラマの責任は重大だ

まじめなプログラマは責任を果たすために1つひとつの命令からの戻り値をチェックするめんどうを引き受けて苦しみますし、不まじめなプログラマは責任を放棄して例外処理をサボってしまうのです。

15.2.2 新しい例外処理の方法

このような従来型の例外処理の問題点を解決するために、Javaをはじめとする新しいプログラミング言語では、例外処理専用の文法としくみが備わっています。

先ほどの「ファイルに『hello!』と書き込む」をJavaで書くと次の図15-7のようになります。やっていることは先ほどのC言語の例と同じで、①ファイルを開く、②ファイルに書く、③ファイルを閉じる、の3つです。

図15-7 Javaでの例外処理の例では本来の処理と例外処理がハッキリ分かれている

このプログラムでは「java.io.FileWriter」という、まだ紹介していないクラスを使っているが、「ファイルに書き込むための API」だと理解してほしい。

この例にあるように、Javaでは例外処理にtryとcatchという2つのブロックを使用します（合わせて**try-catch文**と呼びます）。

tryとcatch、2つのブロックのうち**通常、実行されるのはtryブロック**だけで、

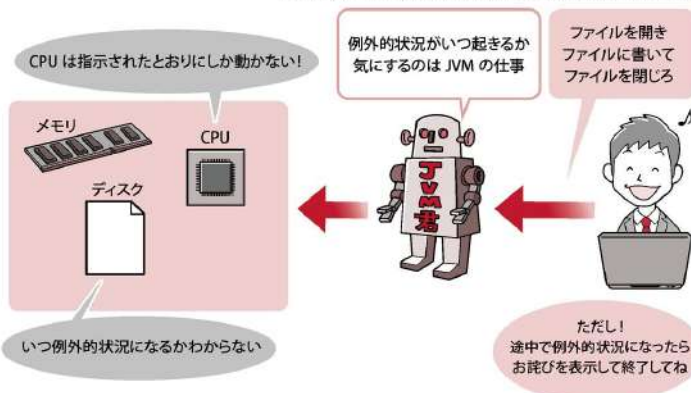
catchブロックの処理は動きません。ただし、**tryブロック内を実行中に例外的状況が発生したことをJVMが検知すると、処理は直ちにcatchブロックに移行**します。つまり、catchブロックの中には「例外的な状況が発生したときに実行される処理」を記述しておくのです。

Javaにおける例外処理の基本パターン

```
try {
    通常実行される文
} catch( … ) {
    例外発生時に実行される文
}
```

見方を変えれば、tryブロックとは「この部分では例外的状況が発生する可能性があるから、その検出を試みながら実行しろ」というプログラマからJVMへの指示ともいえます。

命令を実行するたびに「例外的な状況が発生しているか？」をチェックするめんどうな作業はJVMに任せることで、プログラマが負う責任は軽減されます。

図15-8　JVMが例外を検知したら処理を切り替えてくれる

15.3 例外クラスとその種類

15.3.1 例外を表すクラス

先輩。図 15-7 (p.569) のコードにある catch の直後に書かれた「IOException」って何ですか？

それを理解するために、例外クラスについて紹介しよう。

一口に例外的状況といっても、「ファイルがない」「メモリが足りない」「変数が null」など、さまざまな状況があります。それらを同じものとして扱うと、「発生した例外的状況に応じた処理を行う」ことができません。そこで Java では、発生した例外を区別できるように、**それぞれの例外的状況を表すクラス**が複数、準備されています。

「例外的状況をクラスにした」という意味がわかりません…。

大丈夫、落ち着いて第 II 部を思い出そう。

第 II 部で学んだように、オブジェクト指向は「現実世界の何か」をオブジェクトとして Java 仮想世界で再現したものでした。多くのオブジェクトは、ヒトやモノなど「現実世界で形があるもの」から作り出されることが一般的です。

しかし「現実世界で形がないもの」からオブジェクトを作ることもあります。たとえば、イベント運営会社の「イベント情報管理プログラム」を開発する場合、本来は形があるものと見なさない「イベント」を Event クラス（フィールドとして開催日や主催者を持つ）として作るでしょう。

同様に、「ファイルがなくて困っている状況」や、「null が入っていて困っている状況」など現実世界の例外的状況（想定外の事態）をクラスにしたものが例外クラスです。

ちなみに java.lang.IOException は「ファイルの読み書きなどの入出力ができなくて困っている状況」のためのクラスであり、ほかにも多くの例外クラスが API として定義されています。

図 15-7 は「ファイルが読み書きできない例外的状況に陥ったら、catch ブロックの中身を動かせ」という指示なんですね。

そのとおりだ。詳しくは 15.4 節で説明しよう。

15.3.2　例外の種類

API で提供されている例外クラスは、次の図 15-9 のような継承階層を構成しています。

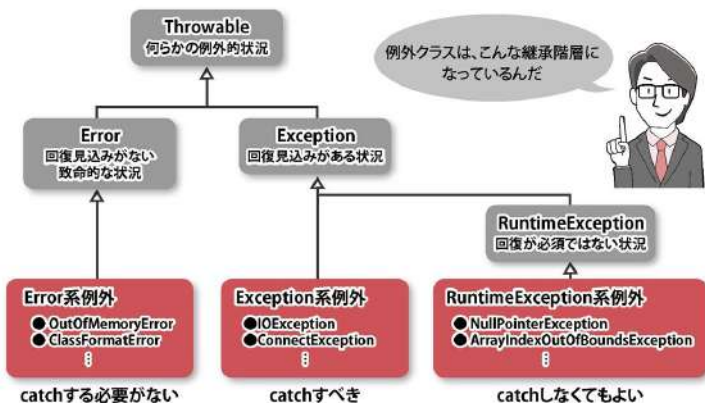



図 15-9　例外クラスの継承階層

① Error 系例外

java.lang.Error の子孫で**回復の見込みがない致命的な状況**を表すクラスです。代表的なものに OutOfMemoryError（メモリ不足）や ClassFormatError（クラスファイルが壊れている）があります。通常、このような状況をキャッチしても打つ手はないため、キャッチする必要はありません。

② Exception 系例外

java.lang.Exception の子孫（RuntimeException の子孫を除く）で、**その発生を十分に想定して対処を考える必要がある例外的状況**を表すクラスです。たとえば、IOException（ファイルなどが読み書きできない）や ConnectException（ネットワークに接続できない）といった状況は、ファイルやネットワークを利用する際に当然、想定しておくべき事態です。

③ RuntimeException 系例外

java.lang.RuntimeException クラスの子孫で、**必ずしも常に発生を想定すべきとまではいえない例外的状況**を表すクラスです。たとえば、NullPointerException（変数が null である）や ArrayIndexOutOfBoundsException（配列の添え字が不正）のように、いちいち想定していると「きりがない」ものが多く含まれます。

15.3.3 チェック例外

APIには多くの例外クラスが定義されていますけど、これだけたくさんの例外的状況が起こる可能性がある、ということですよね…。

めんどうくさいなぁ…。ボクはサボって try-catch とか書かないと思います。

そうは問屋、もとい JVM が卸さないんだよ。

先ほどの３種類ある例外クラスの中で、特に注目してほしいのは②のException系例外です。これは「その発生を十分に想定して対処を考えておく必要がある状況」を表しているのですから、**いざ例外が発生したときに何も対処されないということはあってはならない**はずです。

そのためJavaでは、**Exception系の例外が発生しそうな命令を呼び出す場合、try-catch文を用いて「例外が発生したときの代替処理」を用意しておかないとコンパイルエラー**になります。

リスト 15-1　例外処理を用意していないと…

Main.java

```
1  import java.io.*;
2  public class Main {
3    public static void main(String[] args) {
4      // FileWriterのコンストラクタは、IOExceptionを発生させる
5      // 可能性があります。しかしtry-catchでは囲みません
6      // （失敗時にどうするか、考えていない）。
7      FileWriter fw = new FileWriter("data.txt");
8    }
9  }
```

コンパイル結果

```
Main.java:7: 例外 java.io.IOException は報告されません。スローするにはキャッチまたは、スロー宣言をしなければなりません。
    FileWriter fw = new FileWriter("data.txt");
                    ^
```

そこで次のようにtry-catch文を用いてIOExceptionの発生に備えれば、コンパイルエラーはなくなります。

リスト 15-2　try-catch 文で Exception 系例外の発生に備える

Main.java

```
1  import java.io.*;
```

```
2  public class Main {
3    public static void main(String[] args) {
4      try {
5        FileWriter fw = new FileWriter("data.txt");
6
7
8      } catch (IOException e) {
9        System.out.println("エラーが発生しました。");
10     }
11   }
12 }
```

FileWriter のコンストラクタは、IOException を発生させる可能性がある。

例外的状況になったときに備えて記述された代替処理

このように、Exception 系例外は、例外発生時の対策が用意されているかをコンパイルの時点でチェックされるため、**チェック例外**（checked exception）とも呼ばれます。

必ず try-catch 文を書かないと実行できないなんて、めんどうですよぉ…。

誰かさんみたいに例外処理を書かない人がいるから「サボれないしくみ」になっているのね。

3 つの例外クラスのグループとキャッチの強制

- Error 系例外　try-catch 文でキャッチする必要はない。
- Exception 系例外　try-catch 文でキャッチしないとコンパイルエラー。
- RuntimeException 系例外　try-catch 文でキャッチするかは任意。

15.3.4 発生する例外の調べ方

でも、どの命令を呼んだら、どんなエラーが発生するかなんて想像もつかないし、書きようがないじゃないか。

確かにそうよね…。リスト15-1では当然のように「FileWriterのコンストラクタを呼ぶとIOExceptionを発生させる可能性がある」って書いてあったけど…。

ここまで、FileWriterを用いてファイルにデータを書き込むプログラムを例に解説してきました。しかし、APIに含まれるクラスには、他にも「呼び出すと何らかの例外を発生させる可能性があるメソッド」が数多くあります。

特にチェック例外が起きる可能性のあるメソッドを呼び出す場合はtry-catch文で囲まなければならないので、**「どのメソッドを呼び出したら、どのような例外が発生する可能性があるか」をあらかじめ知っておく必要があります。**

実は「どのクラスの、どのメソッドが、どのような例外を発生させる可能性があるか」という情報は、APIリファレンスに掲載されています。

FileWriter

public **FileWriter**(String fileName)
throws IOException
ファイル名を指定してFileWriterオブジェクトを構築します。

図15-10 FileWriterクラスのコンストラクタ

メソッドやコンストラクタを呼び出した際にException系の例外が発生する可能性がある場合、**引数リストの後に「throws 例外クラス名」と表記されます。**図15-10は、APIリファレンスからFileWriterクラスの解説を抜粋したものですが、「throws IOException」との記載があるので、「FileWriterのコンストラクタを呼び出す（インスタンスを生成する）ときには、IOExceptionをキャッチするtry-catch文が必要になる」と理解すればよいのです。

15.4 例外の発生と例外インスタンス

15.4.1 例外インスタンスの受け渡し

IOException が何かはわかりましたが「catch(IOException e)」のeって何ですか？

その解説をしていなかったね。try-catch 文の構文を振り返りながら e の役割を説明していこう。

ここで再び図 15-7（p.569）のコードにおける catch ブロック部分に注目してください。

```
try {
  ⋮
} catch (IOException e) {
  ⋮
  System.out.println("エラーです。終了します。");
  System.exit(1);
}
```

15.2 節で説明したように、try ブロック実行中は JVM が例外的状況の発生を監視しながらプログラムを実行します。そして、いざ例外が発生すると、JVM は処理を catch ブロックに移行します。このとき JVM は、「プログラムの中のどこで・どのような例外が起きたのか」という**例外的状況の詳細情報が詰め込まれた IOException インスタンスを catch 文で指定された変数 e に代入します。**

図 15-11　例外インスタンスの受け渡し

catch ブロックの中では、この変数 e に格納された詳細情報を取り出して、適切なエラー処理（画面にエラーメッセージとして表示するなど）を行うことができます。

15.4.2 例外インスタンスの利用

例外インスタンスの中には、どんな情報が入っているんですか？

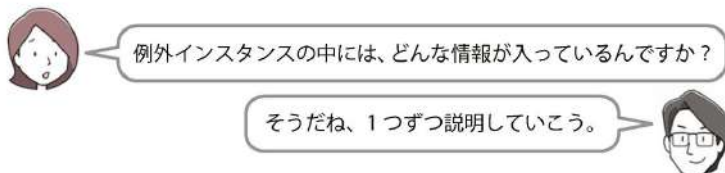
そうだね、１つずつ説明していこう。

例外インスタンスに格納されている詳細情報は、その例外の種類によって異なります。しかし、すべての例外は「例外的情報の解説文」と「スタックトレース」の情報を必ず持っており、それぞれ以下のメソッドで取得と表示ができます。

表 15-2　例外インスタンスが必ず備えているメソッド

メソッド	意味
String getMessage()	例外的状況の解説文（いわゆるエラーメッセージ）を取得する。
void printStackTrace()	スタックトレースの内容を画面に出力する。

図 15-11 のコードでも e.getMessage() がありますね。

そうだね。この場合、もし例外が発生すると画面に「エラー: c:¥data.txt(アクセスが拒否されました)」などが表示されるよ。

スタックトレースとは、「JVM がプログラムのメソッドを、どのような順序で呼び出し、どこで例外が発生したか」という経緯が記録された情報です。「e.printStackTrace();」という文を catch ブロック内に記述すれば、その内容を画面に表示できます。

ところでみなさんは、「スタックトレースがどのような情報であるか」をすでにご存じのはずです。

```
java.io.IOException: data.txt (アクセスが拒否されました。)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:120)
  at java.io.FileInputStream.<init>(FileInputStream.java:79)
  at java.io.FileReader.<init>(FileReader.java:41)
  at Main.main(Main.java:6)
```

実行時エラーが発生したときのエラー画面って、発生した例外のスタックトレースの内容だったんですね。

そうだよ。例外が発生しても try-catch 文でキャッチされなかった場合、JVM がプログラムを強制停止してスタックトレースの内容を画面に表示していたんだ。なお、スタックトレースの詳しい読み方は付録 C を参照してほしい。

15.5 さまざまな catch 構文

15.5.1 try-catch 構文の基本形

例外の種類やしくみについて基礎は理解できたね。ここからは try-catch 文の詳細な構文を解説しながら、例外の捉え方を紹介しよう。

それでは例外処理の基本構文を復習した上で、さまざまな構文のバリエーションを見ていきましょう。まずは基本構文です。

try-catch の基本構文

```
try {
  本来の処理
} catch ( 例外クラス　変数名 ) {
  例外が発生した場合の処理
}
```

なお、例外インスタンスを受け取るための変数名は自由ですが、慣習として e や ex が使用されます。

15.5.2 2 種類以上の例外をキャッチする

前節で紹介した try-catch の基本構文でキャッチできるのは 1 種類の例外だけでした。しかし、図 15-12 のように catch ブロックを複数記述することもできます。

JVMは発生した例外の型に対応するcatchブロックを上から順に検索し、最初にキャッチできたcatchブロックに処理を移します。

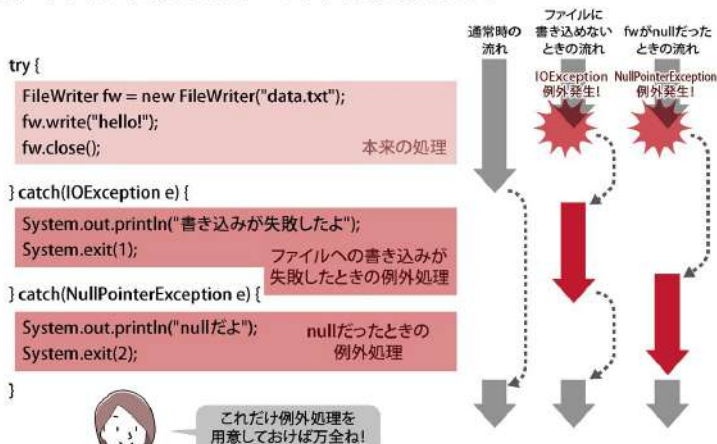



図15-12　複数のcatchブロック

IOExceptionをキャッチしたときは…、NullPointerExceptionをキャッチしたときは…、というif文みたいですね。

なお、図15-12のコードではcatchブロックを2つ記述していますが、IOExceptionとNullPointerExceptionのどちらを捕まえても同じ処理をする場合、「catch(IOException | NullPointerException e) { … }」という記述でcatchブロックを1つにまとめることが可能です。

15.5.3 ザックリと例外をキャッチする方法

catchブロックに指定する例外クラスは、第13章の多態性で学んだ「ザックリ捉えた型」でも構いません。

例外クラスの継承階層図（p.572の図15-9）によれば、IOExceptionもNullPointerExceptionも、ザックリ捉えればどちらもExceptionです。よって次のように記述することで、どちらの例外が発生しても1つのcatchブロックでキャッ

チできます。

リスト 15-3　ザックリと例外を捕捉する

Main.java

```
import java.io.*;
public class Main {
  public static void main(String[] args) {
    try {
      FileWriter fw = new FileWriter("data.txt");
      fw.write("hello!");
      fw.close();
    } catch (Exception e) {
      System.out.println("何らかの例外が発生しました");
    }
  }
}
```

Exception の子孫をどれでもキャッチ

これなら発生する例外の種類が増えても catch ブロックを増やさなくていいから楽ですね。

しかし、どのような種類の例外が発生しても同じように処理するから、大ざっぱな例外処理になってしまうね。

15.5.4 後片付け処理への対応

実はリスト 15-3 には致命的なバグがあるのですが、どこが問題なのかわかりますか？ 答えは「7 行目のファイルを閉じる処理が動かないことがある（ファイルが開いたままになることがある）」です。

ファイルは「開いたら閉じる」のが決まりです。あるプログラムがファイルを開いている間は、ほかのプログラムからそのファイルを使えないことがあるので「閉じ忘れ」はあってはならないのですが、リスト 15-3 では、それが起こってしまいます。

たとえば6行目の「fw.write("hello!");」を実行したときに、偶然ディスクの容量が満杯になりIOExceptionが発生したとしましょう。するとプログラムの処理はcatchブロックに移動してしまうため7行目のfw.close()は実行されず、ファイルは開いたままになってしまいます。

close()は「例外が起きても、起きなくても」必ず実行しなければならない処理ですよね。

リスト15-4　朝香さんが作成したプログラム

Main.java

```
1  import java.io.*;
2  public class Main {
3    public static void main(String[] args) {
4      FileWriter fw = null;
5      try {
6        fw = new FileWriter("data.txt");
7        fw.write("hello!");
8      } catch (IOException e) {
9        System.out.println("エラーです");
10     }
11     fw.close();
12   }
13 }
```

try-catchの後でcloseする

このプログラムなら問題ないように思えます。しかし、もしtryブロックの中でNullPointerExceptionなどが発生した場合、例外はキャッチされないので、ファイルを閉じないままプログラムは強制終了してしまいます。

図 15-13　ファイルを閉じられない不具合の発生

この例で取り上げた「fw.close();」のような後片付け処理は、「例外が発生しても、またはしなくても、たとえ強制終了になるときでも、必ず実行しなければならない処理」です。そして、そのような処理を JVM に確実に実行させるために、次の finally ブロックがあります。

例外発生の如何を問わず必ず処理を実行する

```
try {
  本来の処理
} catch ( 例外クラス　変数名 ) {
  例外が発生した場合の処理
} finally {
  例外があってもなくても必ず実行する処理
}
```

try-catch-finally 構文では、一度 JVM が try ブロックの実行を開始したら、必ず最後に finally ブロックの内容も実行されることが保証されています。

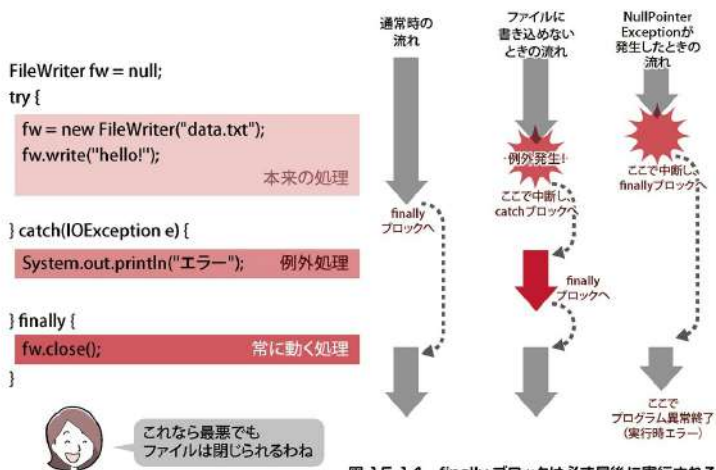



図 15-14　finally ブロックは必ず最後に実行される

開いたファイルを閉じる、開いたデータベースやネットワークとの接続を閉じるなど、**「後片付け処理」には必ず finally** を使います。

必ず finally を使うべき状況

後片付け処理は、必ず finally ブロックに記述する（ファイル・データベース接続・ネットワーク接続など）。

15.5.5 自動的に close() が呼ばれる try-catch 文

finally での後片付けが大事なことはわかるけど、やっぱめんどうだなぁ…。なんかこう、JVM が気をきかせて自動で後片付けしてくれないんですか？

さすがに自動は無理だけど、記述を楽にすることはできるよ。

Java7 以降では、try の直後に丸カッコで囲まれた複数の文を記述することが可能になりました。ここで開かれたファイルやデータベース接続などは、finally ブロックを記述しなくても Java によって自動的に close() メソッドが呼び出されます。

この構文を活用して、図 15-14 のコードを次のように書き直すことができます。

```
try (
  FileWriter fw = new FileWriter("data.txt");
) {
  fw.write("hello!");
} catch(IOException e) {
   ⋮
}
```

try-catch 文を抜ける際に、自動的に close() が呼び出される。finally ブロックの記述は不要

なお、JVM によって自動的にクローズされるのは、java.lang.AutoClosable インタフェースを実装している型に限られます。ファイル操作やデータベース接続、ネットワーク接続に用いる API クラスの多くは、AutoClosable を実装しているため、この節で紹介した方法での簡潔な記述が可能です。

15.6 例外の伝播

15.6.1 main メソッドで例外をキャッチしないと…

ここまでは１つのメソッドに注目して例外処理を解説してきたね。では、main メソッドから呼び出した先のメソッドで例外が発生したらどうなると思う？

うーん、単に、呼び出されたメソッドの中で異常終了するだけなんじゃないかなぁ。

この章の冒頭で解説したように、例外が発生したにも関わらずキャッチしないと実行時エラーとなり、プログラムは強制終了してしまいます。「例外が起きても何もできない＝お手上げ」となり、JVM はスタックトレースを画面に表示して（しかたなく）強制終了するのです。それでは、main メソッドではなく、呼び出した先のメソッドで例外が発生したときはどのような動作になるのでしょうか。次のプログラムを例に考えてみましょう。

- **main メソッドの中では sub() メソッドを呼んでいる。**
- **sub() メソッドの中では subsub() メソッドを呼んでいる。**
- **subsub() メソッドでは、処理中に何らかの例外が発生することがある。**

subsub() メソッド実行時に例外が発生すると、以下のようなことが JVM の中で起きます（図 15-15）。

① まず subsub() メソッドで例外をキャッチしていなければ（try-catch 文がなければ）、「subsub メソッドとしては、この例外的状況に対してお手上げ」となり、呼び出し元の sub メソッドに対応が委ねられます。
② sub() メソッドでもキャッチしなければ、例外の対応は main メソッドに委ねられます。

③ main メソッドで例外をキャッチしなければ強制終了します。

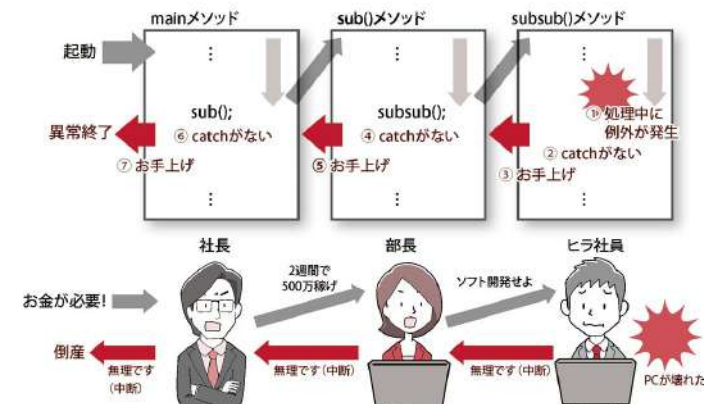



図 15-15　発生した例外はキャッチしないとプログラムが強制終了する

　このように例外はキャッチされない限り、メソッドの呼び出し元まで処理を「たらい回し」にされてしまいます。この現象を**例外の伝播**と呼びます。もちろん呼び出し元の main メソッドや sub() メソッドに catch ブロックが準備されていれば、例外の伝播はそこで止まります（図 15-16）。

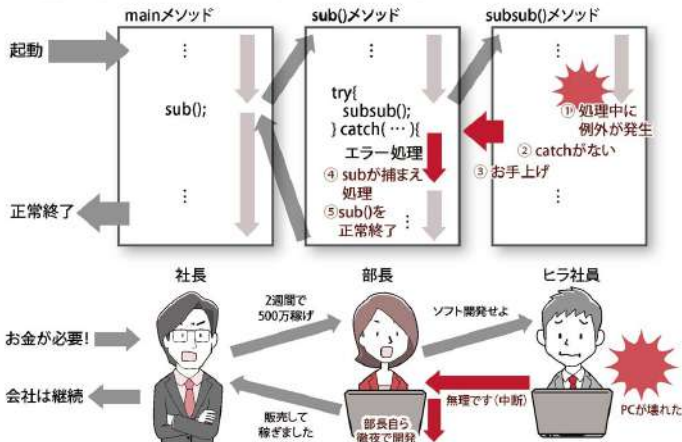



図 15-16　発生した例外は呼び出し元メソッドに処理を委ねる

15.6.2 チェック例外の伝播とスロー宣言

例外の伝播は発生した例外が各メソッドでキャッチされず「お手上げ」になるため起こります。しかし、Exception系例外（チェック例外）はtry-catch文によるキャッチが必須（15.3.3項）なため「お手上げ」になることはなく、基本的に例外の伝播は起こりません。

しかし、メソッドを宣言する際に**スロー宣言**を行うことで、発生するチェック例外を呼び出し元へと伝播させることが許可されます。

スロー宣言による例外伝播の許可

```
アクセス修飾　戻り値　メソッド名(引数リスト)
        throws 例外クラス1, 例外クラス2,…{
  メソッドの処理内容
}
```

たとえば次のように記述します。

```
public static void subsub() throws IOException {
  // IOExceptionが発生する可能性があるが、
  // try-catch文がなくてもOK
  FileWriter fw = new FileWriter("data.txt");
}
```

スロー宣言

あれ？ このsubsub()メソッドって、実行中にチェック例外が起こるはずなのにtry-catchがないですよ。「サボり禁止」でコンパイルエラーになりますよね。

いや、スロー宣言があるときに限って
コンパイルエラーにならないんだよ。

このメソッドの中では FileWriter をインスタンス化しており、チェック例外 IOException が発生する可能性があります。しかし、この例のように**スロー宣言を行っていれば try-catch 文がなくてもコンパイルエラーになりません。**

スロー宣言によるチェック例外の伝播

メソッドを定義する際、自らキャッチしないチェック例外を throws で宣言することができる。このときメソッド内で try-catch 文によるキャッチをしなくてもコンパイルエラーにならない。

なぜスロー宣言をするとコンパイルエラーにならないのでしょうか。それは、スロー宣言とは、そのメソッドが「私はメソッド内でチェック例外が発生しても処理しませんが、**私の呼び出し元が処理します**」と表明する宣言だからです。

その一方、スロー宣言が含まれるメソッドを呼び出す側は「このメソッドを呼び出すと、呼び出し先で発生した例外が処理されずに**自分に伝播してくる可能性がある**」ことを覚悟しなければなりません。

スロー宣言が及ぼす影響

影響①　呼び出される側のメソッド b() は、メソッド内部での～ Exception のキャッチが義務ではなくなる。

影響②　呼び出す側のメソッド a() は、「～ Exception を伝播してくる可能性がある b()」の呼び出しを try-catch 文で囲む義務が生まれる。

※ throws ～ Exception というスロー宣言を伴うメソッド b() を、メソッド a() から呼び出す場合。

チェック例外に対する処理方法についてまとめておきましょう。すべてのメソッドは「チェック例外をどう処理するか」について次の２つの方針のどちらかを採用し、その方針ごとに課せられる義務を果たさなければなりません。

例外処理方針① チェック例外を自分で処理

【この方針の意味】

「私は自分で例外的状況を解決します。例外が発生してもお手上げはせず、呼び出し元に迷惑をかけません」

【この方針を採用することで課せられる義務】

発生する可能性がある、すべてのチェック例外を try-catch 文で処理すること。

例外処理方針② チェック例外を処理せず、呼び出し元に委ねる

【この方針の意味】

「私は自分で例外的状況を解決できません。例外が発生したら、呼び出し元に処理を任せます」

【この方針を採用することで課せられる義務】

メソッド定義にスロー宣言を加え、例外の種類を表明すること。

例外をもみ消さない

```
try {
  ⋮
} catch ( Exception e ) {
}
```

このコードは「発生した例外を捕まえながら、自分では何の処理もせず、上にも報告しない」いわば「不祥事のもみ消し」のようなことをやっています。チェック例外は「何らかの対処がされるべきだから発生している」のですから、もみ消しが重大な不具合につながることは容易に想像できます。ですから空の catch ブロックは極力避けるようにしましょう。もし理由があって「あえて何もしない」場合には、catch ブロックの中にコメントで理由を残しておくとよいでしょう。

例外を発生させる

15.7.1 例外的状況をJVMに報告する

前節までで「例外の発生にどう備えるか」という解説は終わりだ。最後に自分たちで「例外を発生させる方法」を紹介しよう。

　図15-8（p.570）で解説したように、「例外的状況が発生したかどうか」はJVMが監視します。そしてJVMは例外的状況を検知すると処理をcatchブロックに移すのでしたね。実は、この監視をしているJVMに対して、私たち自身が「〜Exceptionという例外的状況になりました」と報告をすることができます。例外的状況が発生したことを報告するためには、次の構文を使います。

例外的状況の報告（例外を投げる）

```
throw 例外インスタンス;
```

※一般的には「throw new 例外クラス名("エラーメッセージ");」となる。

JVMに報告するには、例外の詳細情報を詰め込んだ例外インスタンスを用いるんですね。

そうだよ。throwキーワードを使って、例外インスタンスを監視中のJVMに「投げつける」んだ。

「こんな例外的状況になったよ！ 今すぐ代替策の実施へ移行してください」と例外インスタンスを投げているイメージですね。

監視中のJVMに例外的状況を報告することを、「**例外を投げる**」または「**例外を送出する**」と表現することもあります。例外が投げられるとJVMはそれを検知し、即座にcatchブロックの実行や伝播に処理を移します。

実際に例外を投げているプログラムの例がリスト15-5です。

リスト15-5　例外インスタンスを自分で投げる

Person.java

```
1  public class Person {
2    int age;
3    public void setAge(int age) {
4      if (age < 0) {      // ここで引数をチェック
5        throw new IllegalArgumentException
             ("年齢は正の数を指定すべきです。指定値=" + age);
6      }
7      this.age = age;     // 問題ないなら、フィールドに値をセット
8    }
9  }
```

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Person p = new Person();
4      p.setAge(-128);
5    }
6  }
```

誤った値のセットを試みる→例外発生

実行結果

```
Exception in thread "main" java.lang.IllegalArgumentException:年齢は正の数
を指定すべきです。指定値=-128
  at Person.setAge(Person.java:5)
  at Main.main(Main.java:4)
```

Personクラスの setAge() メソッドでは、引数をフィールド age（年齢）に代入します。しかし age が負の値になることを防ぐため、代入の前に引数を検査しています。もし引数に問題がある場合は、IllegalArgumentException インスタンスを投げることで「引数が異常で処理を継続できない」という例外的状況に陥ったことを JVM に報告します。

setAge() メソッドで発生した例外は、呼び出し元の main メソッドに伝播します。しかし、ここでは例外をキャッチしていないので、最終的に JVM がプログラムを強制停止します。

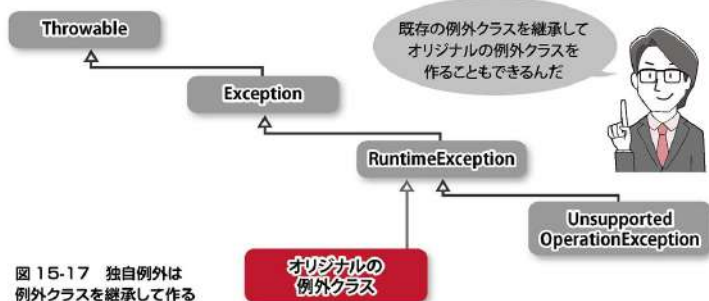
スロー宣言で使う throws と例外的状況の報告に使う throw は、似ているけど、まったく違うものだから気をつけよう。

15.7.2 オリジナル例外クラスの定義

これまで見てきたように、API には IOException や IllegalArgumentException などの多くの例外クラスが備わっています。それら既存の例外クラスを使えば、多くのプログラムは問題なく作成できるでしょう。

しかし「独自の例外的状況」を表す「オリジナルの例外クラス」を使いたくなることもあります。たとえば音楽プレーヤソフトを開発しているのであれば、「対応していない形式のファイルを再生しようとした」などの例外的状況を表すクラス（たとえば、UnsupportedMusicFileException）が欲しくなるかもしれません。

そのような場合は、**既存の例外クラスを継承してオリジナルの例外クラスを作る**ことができます。

図15-17　独自例外は例外クラスを継承して作る

継承元になる例外クラスは、チェック例外を表す Exception や、非チェック例外を表す RuntimeException の他、IOException など実際に何かの状況を表している例外クラスでも構いません。しかし、Throwable や Error の子クラスとしてオリジナル例外を定義することはほとんどないでしょう。

リスト 15-6　オリジナル例外を定義する

UnsupportedMusicFileException.java

```
1  public class UnsupportedMusicFileException extends Exception {
2    // エラーメッセージを受け取るコンストラクタ
3    public UnsupportedMusicFileException(String msg) {
4      super(msg);
5    }
6  }
```

チェック例外にする

リスト 15-7　オリジナル例外を利用する

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      try {
4        // 試験的に例外を発生させる
5        throw new UnsupportedMusicFileException
           ("未対応のファイルです");
6      } catch (Exception e) {
7        e.printStackTrace();
8      }
9    }
10 }
```

本格的で大規模なプログラムを開発するときは、プログラムで想定されるさまざまな例外的状況を思い浮かべ、オリジナルの例外クラスとして作成することで、きめ細かい実行時エラーへの対処が可能になります。

15章

15.8 第15章のまとめ

この章では、次のようなことを学びました。

エラー

- 「文法エラー」、「実行時エラー」、「論理エラー」の 3 種がある。
- 例外処理を行うことで、実行時エラーに対処できる。

例外の種類

- API には、さまざまな例外的状況を表す例外クラスが用意されている。
- 例外クラスは「Error 系」、「Exception 系」、「RuntimeException 系」に大別できる。
- 例外クラスを継承してオリジナルの例外クラスを定義できる。

例外処理

- try-catch文を使用すると、try ブロック内で例外が発生したときに catch ブロックに処理が移る。
- 後片付けの処理は、必ず実行される finally ブロックに記述する。
- Exception 系例外が起こる可能性がある場合は、try-catch 文が必須である。
- スロー宣言を行うことで、例外の処理を呼び出し元に委ねることができる。
- throw 文を使うことで、開発者自ら例外を発生させることができる。

15.9 練習問題

練習 15-1

次のようなプログラムを作成・実行し、実行時エラーを発生させてください。

① String 型変数 s を宣言し、null を代入する。
② s.length() の内容を表示しようとする

練習 15-2

練習 15-1 で作成したコードを修正し、try-catch 文を用いて例外処理してください。その際に例外処理では次の処理を行ってください。

①「NullPointerException 例外を catch しました」と表示する。
②「－－スタックトレース（ここから）－－」と表示する。
③スタックトレースを表示する。
④「－－スタックトレース（ここまで）－－」と表示する。

練習 15-3

Integer.parseInt() メソッドを実行し、文字列 " 三 " の変換結果を int 型変数 i に代入するコードを記述してください。その際に、parseInt() メソッドがどのような例外を発生させる可能性があるかを API リファレンスで調べ、正しく例外処理を記述してください。

練習 15-4

起動直後に IOException を送出して異常終了するようなプログラムを作成してください（ヒント：main() メソッドが「お手上げ」すれば、例外発生時にプログラムが異常終了します）。

15.10 練習問題の解答

練習 15-1 の解答

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String s = null;
4      System.out.println(s.length());
5    }
6  }
```

練習 15-2 の解答

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      try {
4        String s = null;
5        System.out.println(s.length());
6      } catch (NullPointerException e) {
7        System.out.println
             ("NullPointerException例外をcatchしました");
8        System.out.println("――スタックトレース（ここから）――");
9        e.printStackTrace();
10       System.out.println("――スタックトレース（ここまで）――");
11     }
12   }
13 }
```

練習 15-3 の解答

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      try {
4        // APIリファレンスから送出例外を調べる
5        int i = Integer.parseInt("三");
6      } catch (NumberFormatException e) {
7        System.out.println
             ("例外NumberFormatExceptionをcatchしました");
8      }
9    }
10 }
```

練習 15-4 の解答

Main.java

```
1  import java.io.IOException;
2  public class Main {
3    public static void main(String[] args) throws IOException {
4      System.out.println("プログラムが起動しました。");
5      throw new IOException();
6    }
7  }
```

第 16 章

まだまだ広がるJavaの世界

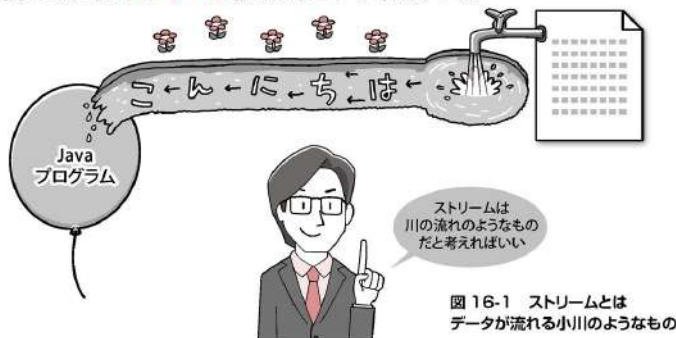
第 14 章以降では重要な Java の API について学んできました。
本書で解説した API は Java 全体から見ればごく一部ですが、
本書を理解した方であれば、自力で API を調べ、
自分のプログラムに取り入れて活用することができるでしょう。
この最終章では読者のみなさんへの「はなむけ」として、さらに高度な
Java プログラミングの世界と可能性を紹介します。

CONTENTS

16.1 ファイルを読み書きする

16.1.1 ストリーム

プログラムからコンピュータの中にあるファイルを読み書きする場合、ファイルの内容をすべて一度に変数へ読み込むことは通常、行いません。なぜなら、ファイルがとても大きかった場合、一度に読み込むとメモリが足りなくなってしまうからです。そこでJavaをはじめとする多くのプログラミング言語では、ファイルを「少しずつ読んだり書いたり」するための機能を備えています。このとき必要となるのが**ストリーム**(stream)という考え方です。

**図16-1　ストリームとは
データが流れる小川のようなもの**

ストリームとは、情報が流れてくる小川のようなものだと想像してください。Javaプログラムは、この小川を通してファイルを読み書きします。たとえばファイルを読み込む場合は、図16-1のように、「小川の上流にあるファイルから1文字ずつ流れてくる」というようなイメージで文字を読み込みます。

16.1.2 ファイルから文字を読み込む

ファイルから文字を1文字ずつ読み込むコードを紹介しましょう(リスト16-1)。テキストファイルから文字を読み込む場合には、java.io.FileReaderを使います。

リスト 16-1

Main.java

```
 1  import java.io.*;
 2  public class Main {
 3    public static void main(String[] args) throws Exception {
 4      String filename = "c:¥¥test.txt";
 5      FileReader fr = new FileReader(filename);
 6      char c1 = (char) fr.read();
 7      char c2 = (char) fr.read();
 8      fr.close();
 9    }
10  }
```

ファイル名をセットする
ファイルを開く
最初の 1 文字を読む
次の 1 文字を読む
ファイルを閉じる

FileReader は「指定されたファイルが源流にある小川」のようなもので、read() メソッドを呼ぶたびに 1 文字ずつ文字を取り出せます。read() の結果が -1 の場合、ファイルを最後まで読み終わったことを意味します。そして読み終わったら最後に必ず close() してファイルを閉じておきます。

なお、FileReader のコンストラクタや read()、close() は IOException を送出する可能性があります。このリスト 16-1 はサンプルですので例外処理を記述していませんが、開発の現場では必ず例外処理を行ってください。

close() は必ず finally の中に書くようにね。その理由がわからない人は第 15 章を読み返してほしい。

16.1.3 ファイルへ文字を書き込む

ファイルに文字や文字列を書き込むには、FileWriter を使います。このクラスは「下流にあるファイルにつながっている小川」のようなもので、ストリームに流した文字がファイルに書き込まれていきます。

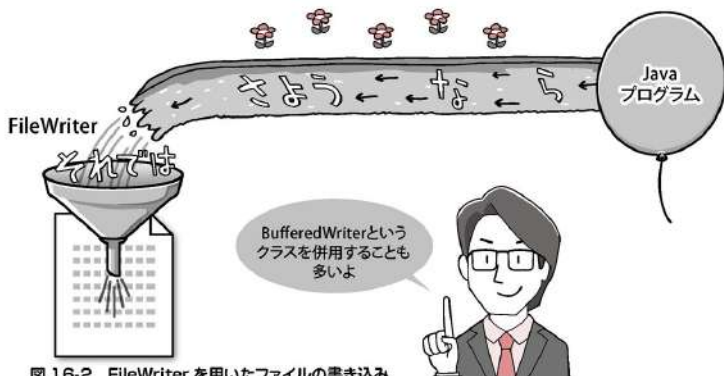



図 16-2　FileWriter を用いたファイルの書き込み

実際に FileWriter を使ってファイルに書き込む例がリスト 16-2 です。

リスト 16-2

Main.java

```
import java.io.*;
public class Main {
  public static void main(String[] args) throws Exception {
    String filename = "c:¥¥test.txt";
    FileWriter fw = new FileWriter(filename);
    fw.write('そ');
    fw.write('れ');
    fw.close();
  }
}
```

- 4〜5行目：ファイルを開く
- 6行目：最初の 1 文字を書く
- 7行目：次の 1 文字を書く
- 8行目：ファイルを閉じる

今回は FileReader と FileWriter の 2 つのクラスだけを紹介しましたが、java.io パッケージにあるさまざまなクラスを利用すると、より高度な入出力処理が可能になります。

16.2 インターネットにアクセスする

16.2.1 Web ページを取得する

従来のプログラミング言語では、インターネット上の Web ページの内容を取得するために、ネットワークプログラミングに関する多くの知識とプログラミング作業が必要でした。しかし、Java では java.net パッケージのクラスを使うことで、同じ機能のプログラムを、ほんの数行で書くことができます（リスト 16-3）。

リスト 16-3

Main.java

```
1  import java.io.InputStream;
2  import java.net.URL;
3  public class Main {
4    public static void main(String[] args) throws Exception {
5      URL u = new URL("http://www.impressjapan.jp/");
6      InputStream is = u.openStream();
7      int i = is.read();
8      while (i != -1) {
9        char c = (char) i;
10       System.out.print(c);
11       i = is.read();
12     }
13   }
14 }
```

- 6行目：ネットへ接続
- 8行目：ページの終わりまで繰り返す
- 10行目：読んだ内容を画面に表示

実行結果

```
<!DOCTYPE html PUBLIC …>
```

インプレスのサイト内容

16章

```
<html xmlns="http://www.w3.org/1999/xhtml"><head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Impress</title>
 …
```

リスト 16-3 を実行すると、Web ページを構成している HTML のテキストが画面に表示されます。このプログラムのポイントは 5 行目で java.net.URL クラスのインスタンスを生み出し、openStream() メソッドを呼び出している部分です。このメソッドを呼ぶと、インターネット上のページを上流に持つストリームが取得できますので、1 文字ずつ読みながら画面に出力しています。

図 16-3　openStream() メソッドにより
Web ページを上流に持つストリームが構成される

さまざまなモノにつながるストリーム

Java のストリームはファイルや Web ページ以外にもさまざまなモノを接続できます。実は今までにも私たちはストリームを何度も使ってきました。それは下流が画面につながっている小川である「System.out」と、上流がキーボードにつながっている「System.in」です。System.out.println() とは、画面につながっている小川に情報を流す命令だったのです。

16.3 データベースを操作する

16.3.1 データベースと SQL

データベースに情報を入れたり出したりするのにも、ストリームが使えるんですか？

残念だが、データベースの操作にストリームは使えない。その代わりに SQL という専用の指示をデータベースとやりとりするよ。

データベースとは、データを整理して格納したり、高速に取り出したりするためのソフトウェアとデータの集合体を指します。

一般的なデータベースの中には多くの「表」があり、その表の中の値を取得したり書き換えたりして利用します。実際に表の中身を読んだり書いたりするためには、「SQL」というデータベース専用の言語でデータベースに指示を送る必要があります。

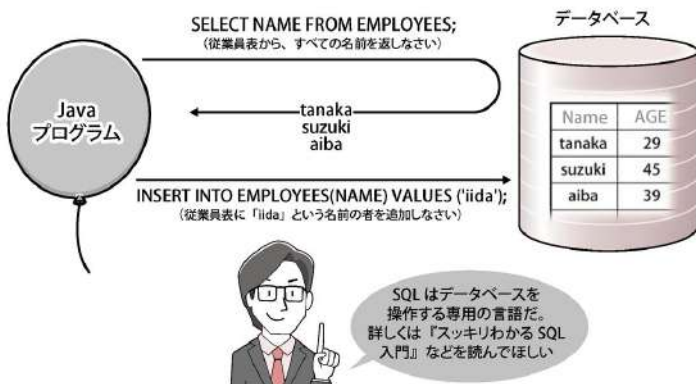



図 16-4　データベースに SQL 文を送り、表を読み書きする

図16-4の中の「SELECT ～」や「INSERT ～」がSQLの命令なんですね。

そうだよ。SQL文をデータベースに送れば、その指示に従ってデータを処理してくれるんだ。

Javaではjava.sqlパッケージのクラスを用いることで、データベースにSQL文を送ることができます。詳細は割愛しますが、リスト16-4のサンプルコードを眺めて、「データベースアクセスも決して難しくない」ということを実感してみてください。

リスト16-4

Main.java

```
1  import java.sql.*;
2  public class Main {
3    public static void main(String[] args) throws Exception {
4  Class.forName("org.h2.Driver");
5      String dburl = "jdbc:h2:~/test"
6                              接続先DBを指定
7      String sql = "INSERT INTO EMPLOYEES(name) VALUES('iida')";
8      Connection conn = DriverManager.getConnection(dburl);
9                                         DBに接続
10     conn.createStatement().executeUpdate(sql);
11                            SQLを送信
12     conn.close();                       DB接続を閉じる
13   }
14 }
```

16.4 ウィンドウアプリケーションを作る

16.4.1 CUI と GUI

湊くんの RPG はおもしろいけど、表示が文字だけでカワイくないのが残念ね…。

少しガンバれば、ウィンドウを持ったグラフィカルなプログラムも Java で作ることができるんだよ。

プログラムの見た目・操作性のことを**ユーザーインタフェース**（UI：User Interface）といいます。その中でも本書で開発してきたような文字ベースのユーザーインタフェースは **CUI**（Character User Interface）と呼ばれます。

一方、Windows や Mac などのパソコンで使われる Web ブラウザやメールソフトのような「窓枠があってグラフィカルな表示とマウスを使って操作できるソフトウェア」は **GUI**（Graphical User Interface）と呼ばれます。

CUIプログラム

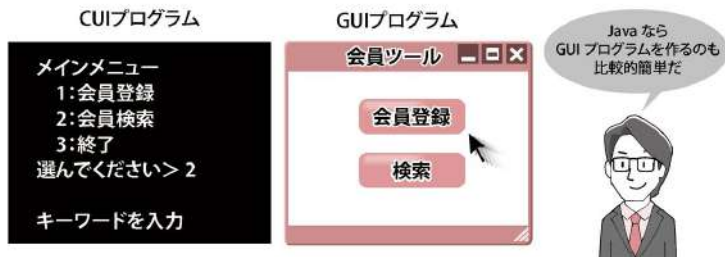
```
メインメニュー
  1:会員登録
  2:会員検索
  3:終了
選んでください> 2

キーワードを入力
```

GUIプログラム

図 16-5　CUI と GUI の違い

java.awt と javax.swing パッケージには、GUI 開発に用いるボタンや入力ボックスなど、さまざまな部品がクラスとして提供されています。紙幅に制限があり

ますので、本格的な GUI プログラムの作り方の解説は専門書に譲り、ここでは簡単なサンプルコードの紹介にとどめます（リスト 16-5）。

しかし、Java を使えば Windows、Mac、Linux、そのほか Java が動作するコンピュータであれば、どれでも同じように動作するウィンドウアプリケーションが作れるということを、ぜひ知っておいてください。

リスト 16-5

Main.java

```
import java.awt.FlowLayout;
import javax.swing.*;
public class Main {
  public static void main(String[] args) {
    JFrame frame = new JFrame("はじめてのGUI");
    JLabel label = new JLabel("Hello World!!");
    JButton button = new JButton("押してね");
    frame.getContentPane().setLayout(new FlowLayout());
    frame.getContentPane().add(label);
    frame.getContentPane().add(button);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 100);
    frame.setVisible(true);
  }
}
```

図 16-6　リスト 16-5 の実行結果

16.5 スマートフォンのアプリを作る

16.5.1 携帯端末で動く Java

菅原さん！ Java でスマートフォン用のアプリケーションも作れるって本当ですか！？

そうだよ。湊くんの RPG がスマートフォンでも動いたら、遊んでくれる人は増えるんじゃないかな。

最近のスマートフォンをはじめとする携帯端末には、Java で開発したアプリケーションを動かせるものがあります。たとえば NTT ドコモの「i アプリ」や、Android 端末用のアプリケーションは Java で開発します。

ただし、Windows や Mac 用に開発した Java プログラムが、そのまま携帯端末で動くわけではありません。携帯端末を開発しているメーカー各社から、それぞれの機種専用の「プログラミングで利用するクラス」が SDK（Software Development Kit）として提供されており、それらの専用クラスを用いてプログラミングを行う必要があります。

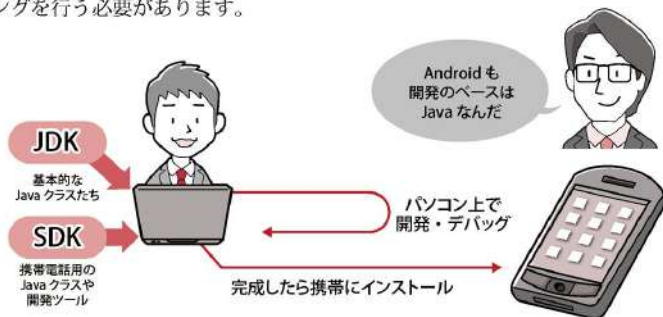



図 16-7　Java で携帯端末用アプリケーションも開発できる

通常、携帯端末用のアプリケーション開発には、SDK に同梱される「エミュレータ」と呼ばれる携帯端末の動作を擬似的に再現するパソコン用ソフトを使います。開発中のプログラムをエミュレータで実行して動作を確認し、プログラムが完成したら実際に携帯端末にインストールして利用するわけです。

16.5.2 Android 端末用の HelloWorld

参考までに Android 端末で「Hello Android」と画面に表示する Java コードを掲載します（注：このリスト 16-6 をコンパイルするには Android 用の SDK が必要です）。

リスト 16-6

HelloAndroid.java

```
1  package my.packages;
2  import android.app.Activity;
3  import android.os.Bundle;
4  import android.widget.TextView;
5  public class HelloAndroid extends Activity {
6    public void onCreate(Bundle savedInstanceState) {
7      super.onCreate(savedInstanceState);
8      TextView tv = new TextView(this);
9      tv.setText("Hello Android");
10     setContentView(tv);
11   }
12 }
```

実際には Android 端末用の SDK だけでなく、さまざまな開発ツールのセットアップや、Java コード以外の開発も必要です。詳しくは Android 端末用のアプリケーション開発に関する解説書を参照してください。

16.6 Web サーバで動く Java

16.6.1 Web アプリケーションとは

インターネットが普及し始めた当時の Web サイトは、ブラウザで Web サーバにアクセスして格納されているページ内容を読むためだけのものでした。しかし、最近の Web サイトには、利用者が入力した情報に応じてサーバ側で必要な処理が実行され、画面の表示内容が変化するようなものが多くあります。

たとえば新幹線の指定席を予約できる Web サイトでは、希望の乗車時間と出発駅・目的駅を入力すれば、データベースから新幹線のダイヤと空席情報を検索して結果を表示してくれます。さらに予約ボタンをクリックすれば、予約情報が鉄道会社のデータベースに登録され、席の予約もできます（図 16-8）。

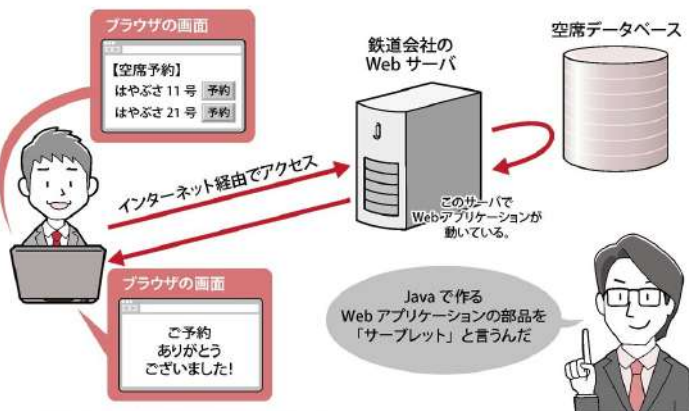



図 16-8　新幹線の予約 Web アプリケーション

この新幹線の Web サイトのように、利用者がブラウザから入力した情報をサーバ側のプログラムで処理するしくみを備えた Web サイトを **Web アプリケーション**（Web Application）といいます。検索サイトやショッピングサイト、あるいは

SNS など、読者の方々も数多くの Web アプリケーションを利用したことがあるはずです。

16.6.2 Java で作る Web アプリケーション

Web アプリケーションは、さまざまなプログラミング言語を使って開発できますが、Java を使って Web アプリケーションを開発する場合には**サーブレット**(Servlet)というクラスを開発します。

サーブレットを開発、動作させるためにはさまざまな準備が必要ですが(参考書籍:『スッキリわかるサーブレット＆ JSP 入門』(インプレス)など)、ここでは「アクセスされたら現在時刻を取得して Web ページとして返す」簡単なサーブレット(HelloServlet)のサンプルコードを紹介します(注:次のリスト 16-7 をコンパイル・実行するにはサーブレットの開発環境が必要です)。

リスト 16-7

HelloServlet.java

```
1  import java.io.*;
2  import java.util.Date;
3  import javax.servlet.http.*;
4  @WebServlet("/HelloServlet")
5  public class HelloServlet extends HttpServlet {
6    protected void doGet(HttpServletRequest req,
         HttpServletResponse res) throws IOException {
7      Date d = new Date();                          ← 現在日付を取得
8      Writer w = res.getWriter();
9      w.write("<html><body>");
10     w.write("Today is " + d.toString());          ← 現在日付を出力
11     w.write("</body></html>");
12   }
13 }
```

ソースコードができたらコンパイルし、いくつかの設定ファイルなどとともに、Java を動かせる機能がある Web サーバに登録します。

たとえば dokojava.jp サーバに、この HelloServlet を登録したとすると、世界中どこからでも「http://dokojava.jp/HelloServlet」というアドレスにブラウザでアクセスすることで現在日付を表示させることができます。

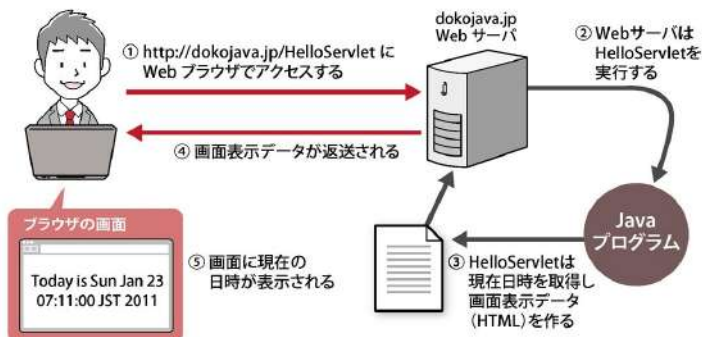



図 16-9　リスト 16-7 の実行の流れ

Java をさらに学んでいけば、もっと多くのいろんなプログラムが作れるようになるんですね。

まだまだ学べることはたくさんあると思うと、なんだか楽しみです。

そうだね。初心者を卒業した 2 人なら大丈夫だから、これからも試行錯誤しながら Java プログラミングを楽しんでほしい。

はい！

さらなる高みを目指して——

湊くんの成長の旅は続きます

Java プログラマ（本書を学習し終えた現在）

オブジェクト指向を活用して、簡単なコマンドラインアプリケーションを開発できる！

Java エンジニア

各種 API 命令・設計技法・ツール等を駆使し、チームでアプリケーションを開発できる！

Java + DB エンジニア

SQL やデータ設計の知識も活用し、DB を利用した本格アプリケーションを開発できる！

Web アプリエンジニア

SNS やショッピングサイトのような Web ブラウザで動くアプリケーションを設計・開発できる！

付録 B

Eclipse による開発

システム開発の現場では、より効率的な開発作業を実現するために
高度な開発ツールを活用します。
付録 B では、その代表的な存在とも言える Java の統合開発環境
「Eclipse」を用いた開発手順を紹介します。
ぜひ、今後の開発のために役立ててください。
なお、利用する Eclipse のバージョン等によって、
掲載の画面写真が実際のものと異なることがあります。

CONTENTS

B.1 Eclipse の導入

B.1.1 統合開発環境とは

付録 A に記載したとおり、JDK を用いた開発では、テキストエディタ、コンパイラ (javac)、インタプリタ (java) の 3 つのツールを利用することが基本です。しかし開発に慣れるにつれ、これら 3 つを繰り返し利用することが手間に感じられるようになるでしょう。

実際の開発現場では**統合開発環境**(IDE : Integrated Development Environment) と呼ばれる開発ツールを利用することが一般的です。統合開発環境は、エディタ、コンパイラ、インタプリタのすべてを内蔵した開発用ソフトウェアであり、1 つの画面で、ソースコードの編集からコンパイル、実行に至るまで、開発作業をひととおり行うことができます。また、ソースコードを見やすくカラーリングしたり、キー入力を支援したり、デバッグを手助けする機能なども豊富に備わっていることが一般的です。代表的な製品としては、Eclipse や NetBeans がよく知られています。

図 B-1　統合開発環境 Eclipse の画面

B.1.2 Pleiades のインストール

Eclipse は無償で公開されており、公式 Web サイト（http://eclipse.org）からダウンロードできます。ただし、公式サイトには英語版しかありません。そこで、Eclipse にさまざまな拡張機能を追加し、さらに日本語化したインストールパッケージである Pleiades をサイト（http://mergedoc.sourceforge.jp）からダウンロードして利用することをお勧めします。

詳細なインストール方法は「http://devnote.jp/pleiades」を参照してください。

B.1.3 Eclipse の起動

Eclipse を起動する手順を紹介します。

① 「< Pleiades インストールディレクトリ >¥eclipse¥eclipse.exe」をダブルクリックします。

② 開いた画面で「OK」ボタンを押します（図 B-2）。ここに出てくる「ワークスペース」とは Eclipse のプロジェクトを保存する場所です。プロジェクトについては後述しますが、作成するアプリケーションの保存場所と考えればよいでしょう。初期値の「../workspace」は eclipse ディレクトリの 1 つ上の階層にある workspace ディレクトリ、つまり「< Pleiades インストールディレクトリ >¥workspace」のことです。本書ではワークスペースは変更しませんが、任意の場所に変更しても構いません。

③ Eclipse の画面が表示されます。

図 B-2　ワークスペースの選択

B.1.4 Java パースペクティブの選択

Eclipseの画面は、「エディタ」や「ビュー」と呼ばれる複数のウィンドウで構成されています。表示するビューとその配置の設定を**パースペクティブ**といい、現在選択中のものが画面右上隅に表示されます。

デフォルトではPC向けのJavaアプリケーション開発用のパースペクティブ「Java」が選択されています。本書で取り扱うプログラムの開発には、このパースペクティブを利用します。

なお、開発中に誤って別のパースペクティブに切り替えてしまうことがあります。このような場合は、次の操作を行って「Java」パースペクティブを選び直すようにしてください。

① ウィンドウメニュー→「パースペクティブを開く」→「その他」を選択します。

② 次の画面で「Java」を選択して「OK」ボタンを押します（図B-3）。なお、「Java」と「Java（デフォルト）」がある場合、どちらでも構いません。

図B-3 パースペクティブの選択画面

なお、Javaパースペクティブでは、デフォルトで「パッケージ・エクスプローラビュー」「アウトラインビュー」「問題ビュー」「エディタ」等が表示されるようになっていますが、自分の好みに合わせてカスタマイズ可能です。各ビューやエディタのタブの「×」アイコンをクリックすれば非表示にすることができるほか、タブをドラッグして配置を変更できます。

B.2 Eclipse による開発手順

B.2.1 Java プロジェクトの作成

Eclipse では、開発するアプリケーションごとに**プロジェクト**と言われる管理単位を作り、その中にソースコードを作成していきます。プロジェクトにはさまざまな種類のものがありますが、Java のプログラムを開発するためには、まず次の手順で「Java プロジェクト」を作成します。

① ファイルメニュー→「新規」→「Java プロジェクト」を選択します（図 B-4）。パースペクティブを「Java」にしていないと「Java プロジェクト」が表示されないので注意してください。

② 開いた画面で「プロジェクト名」を入力して「完了」ボタンを押します（図 B-5）。プロジェクト名は自由に決めて構いませんが、半角英数字やアンダースコアを用いた、わかりやすいもの（例：List01_01 や SukkiriRPG など）をお薦めします。

③ 作成した Java プロジェクトはパッケージ・エクスプローラビューに表示されます。ダブルクリックすると、内容を展開することができます（図 B-6）。

図 B-4　Java プロジェクトの新規作成操作

図 B-5　新規 Java プロジェクトダイアログへの入力

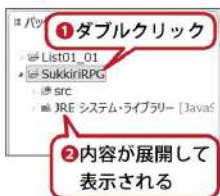



図 B-6　新規作成されたプロジェクト

B.2.2　クラスの作成と編集

Eclipse のプロジェクト内では、次のような手順でクラス（ソースファイル）を新規作成します。

① パッケージ・エクスプローラビューで、クラスを作成したい Java プロジェクトを選択→「右クリック」→「新規」→「クラス」を選択します（図 B-7）。
② 開いた画面で、クラスが所属するパッケージとクラス名を指定し、「完了」ボタンを押します（図 B-8）。
③「src」内にクラスが作成され、エディタに作成したクラスの内容が表示されます。
④ エディタでソースコードを編集し、CTRL+S で保存します。

なお、一度プロジェクト内に作成したクラスは、パッケージ・エクスプローラビューでダブルクリックすることで、いつでも編集することができます。

図 B-7　クラスの新規作成操作

図 B-8　新規 Java クラスダイアログへの入力

本書のソースコードを入力する場合の注意点

Eclipse では、1 つのプロジェクト内に同じ名前のクラスを複数作成することはできません。そのため、本書のソースコードを入力しながら学習する際に困ることがあります。たとえば、リスト 3-1 の Main.java を作成後、リスト 3-2 の Main.java を作成しようとすると「既に Main.java が存在している」旨のエラーが表示され、クラスを新規作成できません。

このような場合、新たに別の Java プロジェクトを作成し、その中にクラスを作成するようにしてください。もしそのクラスが過去のリストで作成したクラスを利用するような場合、それらクラスのソースファイルも新しいプロジェクトにコピーしてください。

B.2.3 コンパイル

Eclipse には Java のコンパイラが含まれており、エディタでコードを書いたり、上書き保存したりすると自動的にコンパイルが行われます。その際にコン

パイルエラーがあると、行の先頭にエラーを表す赤色のマークと、コンパイルエラーが起きた箇所に赤色の波線が表示されます（図 B-9 上）。また、エラーほど致命的ではないものについては、警告（黄色のマークと波線）が表示されます（図 B-9 下）。

```
*Hero.java
1
2 public class Hero {
3     init hp;
4
5     public Hero(int hp) {
6         hp = hp;
7     }
8 }
9
```

コンパイルエラー（赤色波線）

警告（黄色波線）

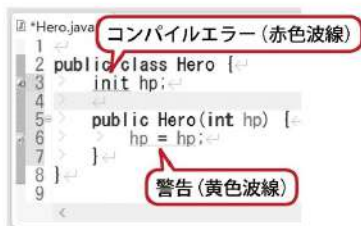



図 B-9　コンパイルエラーおよび警告の発生

B.2.4 デバッグ

　ソースコードを書いている最中に一時的に表示されるコンパイルエラーは気にする必要はありませんが、保存してもコンパイルエラーが表示されている場合は解決する必要があります。エラーのマークの上にマウスポインタを重ねるとエラーメッセージが表示されるので、それを手がかりに原因を探り解決を行います。

　Eclipse には「エラー修正支援機能」があり、エラーのマークをクリックするか、赤の波線の上にマウスポンタを重ねると Eclipse が修正方法の候補をいくつか表示します。その中から 1 つを選択すると、自動でその処理が行われエラーが解消されます。

　ただし、適当に修正候補を選択しただけでは、単にエラーが消えただけで根本的な解決になっていなかったり、余計にエラーを増やしてしまうことがあります。エラー発生の理由を理解しないまま、Eclipse が提示する修正候補を適当に選択するのはやめましょう（付録 C の C.1.1 項のコツ 2 を参照）。

　なお、警告については修正することが必須ではありませんが、メッセージを読んで警告が出ている理由を理解しておくことは重要です。

B.2.5 プログラムの実行

main メソッドを持つクラスは、次の手順で実行することができます。

① 実行したいファイルを選択→「右クリック」→「実行」→「Java アプリケーション」を選択します（図 B-10）。なお、この手順は、画面上部にある緑色の再生ボタンを押すことでも実行できます。

② コンソールビューに実行結果が表示されます（図 B-11）。また java.util.Scanner クラスなどを使ったキーボード入力を受け付けるプログラムの場合、利用者はコンソールビュー内でキー入力を行ってください。

③ 通常、プログラムの実行は自動的に終了しますが、無限ループなどに突入して終わらなくなってしまった場合や、途中で実行を中断したい場合、コンソールビューにある赤い四角のアイコン（図 B-11 右上）をクリックして強制終了させることができます。

図 B-10　プログラムの実行操作

図 B-11　出力結果の表示と実行停止ボタン

B.2.6 JAR ファイルの利用

第 6 章では、開発中のプログラムから JAR ファイルの中にあるクラスを利用するために行う、クラスパスの設定について学びました。具体的には、OS の環境変数や -cp オプションで、JAR ファイルをクラスパスに登録します。

Eclipse を用いた開発では、上記のような作業は不要です。その代わり、以下のような手順で、JAR を**ビルド・パス**に追加します。

① **利用したい JAR ファイルをドラッグ&ドロップし、目的の Eclipse のプロジェクトフォルダ内にコピーします。**
② **Eclipse プロジェクト内の JAR ファイルを右クリック→「ビルド・パス」→「ビルド・パスに追加」を選びます(図 B-12)。**

図 B-12　ビルド・パスへの JAR ファイルの追加

便利なショートカットキー

Eclipse では、ソースコードの編集中に簡単な操作で便利な機能を呼び出せるショートカットキーが用意されています。中でもお薦めのものを紹介しましょう。

ショートカットキー	機能
CTRL + S	現在編集中のファイルを保存する
ALT + SHIFT + X → J	現在のファイルを実行する
CTRL + SPACE	入力候補を表示する
sysout と入力して CTRL + SPACE	System.out.println(); を入力する
CTRL + SHIFT + O	自動的に import 文を追加する
CTRL + SHIFT + F	編集中のソースコードを整形する

付録 C

エラー解決・虎の巻

プログラミングをしていると、思いどおりに動かないことやエラーに悩まされることが少なくありません。幸い「エラーを解決する方法」にも、「エラーをすぐ解決できるプログラマになる方法」にもコツがあります。
この付録では、エラー解決に関するコツと
エラーメッセージの読み方を解説した上で、
困ったときの状況に応じた対応方法をご紹介します。

CONTENTS

C.1 エラーとの上手なつきあい方

C.1.1 エラーを解決できるようになる 3 つのコツ

Java プログラミングを始めて間もないうちは、開発したプログラムが思うようように動かないことも多いでしょう。ささいなエラーの解決に長い時間を要することもあるかもしれませんが、誰もが通る道ですから自信をなくす必要はありません。しかし、その「誰もが通る道」を可能な限り効率よく通り、「エラーをすばやく解決できるプログラマ」になれたら理想的です。

幸いにも、「エラーをすばやく解決する方法」にはコツがあります。この節ではそのコツを、さらに次節 C.2 以降では状況別のエラー対応方法を紹介しましょう。

コツ 1：エラーメッセージを逃げずに読む

エラーが出ると、エラーメッセージをきちんと読まずに「何が悪かったのだろう、どこが悪いのだろう？」と思いつきでソースコードを書き換え始める人がいます。

しかし**「何が悪いのか、どこが悪いのかという情報」は、エラーメッセージに書いてあります。**その貴重な手がかりを読まないのは「目隠しをして探し物をする」のも同然です。上級者でも難しい「ノーヒント状態でのエラー解決」を、初心者ができるはずがありません。

メッセージが英語、あるいは不親切な日本語であったとしても、エラーメッセージはきちんと読みましょう。特に英語の辞書を引く手間を惜しまないでください。辞書を引くための 1 分の時間で、悩む時間が何時間も減ることもあります。

コツ 2：原因を理解した上で修正する

「なぜ今回のエラーが発生したか」という原因を理解しないままエラーを修正してはいけません。また次回、同じエラーに悩まされるだけです。理解に 1 時

間かかるとしても、二度と同じエラーに悩まされなくなるほうが時間の節約になるでしょう。特に、原因を理解していなくても表面的にエラーを消せてしまう、開発ツールや統合開発環境の「エラー修正支援機能」には注意が必要です。初心者のうちはできるだけ、この機能を使わないようにしましょう。

コツ3：エラーと試行錯誤をチャンスと考える

熟練した開発者がすばやくエラーを解決できるのは、Javaの文法に精通しているからという理由だけではありません。頭の中に「エラーを起こした失敗経験と、それを解決した成功経験」の記憶の引き出しをたくさん持っている、つまり、「**似たようなエラーで悩んだ経験があるから**」なのです。

このことが示すように、エラー解決上達のためには、「たくさんのエラーに出会い、試行錯誤し、引き出しを1つずつ増やすこと」が不可欠です。つまり、誰もが避けたいと思う**新しいエラーに直面して試行錯誤している時間こそ、自分が最も成長しているとき**なのです。深く悩むときや切羽詰まるときもあるでしょうが、「自分は今、成長している」と考え、前向きに試行錯誤してください。

これら3つのコツの中で、基本であり最も重要なのが「エラーメッセージをきちんと読むこと」です。しかし、「そもそもエラーメッセージの読み方がわからない」という方もいるでしょう。

そこで、次項からはエラーメッセージの読み方を紹介します。

C.1.2 コンパイルエラーの読み方

不具合のあるソースコードをコンパイルすると、次のようにコンパイルエラーが表示されます。

この「エラーの発生場所（ファイル名：行番号で示されている）」と「エラーメッセージ本文」の2つを必ず読み、原因を推測し、修正を行います。

なお、コンパイルエラーが一度に2つ以上表示された場合、上から1つずつ着実に修正することが重要です。「最初のエラーの対応方法がわからないので、とりあえず無視して次のエラーを先に片付けよう」という取り組み方はお薦めしません。なぜなら、最初のエラーが原因で、2番目のエラーが誘発されていることが少なくないからです。このような場合、最初のエラーを修正すれば2番目以降のエラーも自動的に消えます。逆に、2番目以降のエラーを先に解決することが非常に難しい（または不可能な）場合もあります。

C.1.3 スタックトレースの読み方

プログラム実行中に例外が発生し、最後までcatchされないと、JVMが次のようなスタックトレースを出力して強制終了します。

```
> java Main ⏎
Exception in … java.lang.NumberFormatException: null
  at java.lang.Integer.parseInt(Unknown Source)
  at java.lang.Integer.parseInt(Unknown Source)
  at Sub.process(Sub.java:3)
  at Main.caller(Main.java:28)
  at Main.main(Main.java:6)
```


スタックトレースの2行目以降は、下から順に、どのようなメソッドを呼び出して例外が発生したかという経緯を示しています（下から上に向かって読んでいきます）。今回の例では、「Mainクラスのmain()の内部でMainクラスのcaller()を呼び、その内部でSubクラスのprocess()を呼び、その内部でIntegerクラスのparseInt()を呼び、その内部でIntegerクラスのparseInt()を呼んだところ例外が発生した」ことがわかります。

なお、「atクラス名.メソッド名」の右隣には、ソースファイル名とエラーが発生した行番号が表示されます。ソースコードがない一部のAPIなどは、「Unknown Source」と表示されます。

さて、このスタックトレースからエラーの原因を探るには、まず先頭行を見

て発生した例外の種類と詳細情報を確認し、「何が起きたか」を把握します。今回の例では、NumberFormatException とあるため、「数字の形式がおかしい（null であった）」と推測できます。

次に、「どこで起きたか」を把握するために、次の行（「at」から始まる最初の行）を見ます。この行に記述があるクラスおよびメソッド内で今回の例外が発生しており、「例外の直接原因となった場所」がこれだとわかります。

この行のソースコードを確認し、例外がなぜ発生したかが判明すれば、それを修正します。しかし、その行を見ても問題が見つからなかった場合や、その行が API のメソッドである場合は、スタックトレースの次の行を読みます。

たとえば今回の例では、エラーの直接原因は「java.lang.Integer クラスの parseInt メソッド内」です。しかし API として準備されているメソッドにバグがあるとは考えにくく、「java.lang.Integer.parseInt() の**呼び出し方が悪かったのではないか**」と仮定して、その呼び出し元メソッド（スタックトレースの次の行）を読みます。

parseInt メソッドの行を 2 つ下り、「Sub クラスの process メソッド」を例外の間接原因として着目します。このクラスは API のクラスではなく、自分が開発したクラスであるため、誤ったコードが含まれている可能性が比較的高いといえます。Sub.java の 3 行目を確認し、例外がなぜ発生したかが判明すれば、それを修正します。そのコードに問題がなければ、呼び出し元メソッドのコードに誤りがないか検証する作業を繰り返します。

C.2 トラブルシューティング

Java のセットアップができない

C.2.1 Java（JDK）のインストール方法がわからない

症状 一般書籍やサイトなどに掲載されている JDK のインストール方法を参照しながらセットアップを試みましたが、紹介されている画面と、実際にダウンロードサイトにアクセスしたときの画面が異なるため、わからなくなってしまいました。

原因 Java 公式サイトの画面デザインやダウンロードページのアドレスは比較的短期間で変更されることがあります。よって、少し古い解説を参考にすると手順どおりに実施できないことがあります。

対応 本書では最新の JDK セットアップ手順を「http://dokojava.jp/jdk」で公開しています。画面の指示に従って一歩ずつダウンロードやセットアップを行うことができます。

参照 dokojava の Web サイト

C.2.2 JDK がインストールされているフォルダがわからない

症状 JDK をダウンロードしてインストールしましたが、パソコンのどのフォルダにインストールされたかわからなくなりました。

原因 JDK のインストーラー画面で特に指定をしない場合、標準の場所にインストールされます。

対応 Windows の場合、標準では「C:¥Program Files¥Java」フォルダの中に「jdk ～」というフォルダ名でインストールされます。もし見つからない場合や、Windows 以外の場合、OS の「ファイル検索」機能を使って「javac.exe」を検索してください。javac.exe が含まれているフォルダの親フォルダが JDK のフォルダです。

参照 dokojava の Web サイト

C.2.3 環境変数の設定がわからない

症状 JDK をダウンロードしてインストールしましたが、その後の JAVA_HOME や PATH といった環境変数の設定方法がよくわかりません。

原因 Java 開発のためには、JDK のインストールだけではなく、その後の環境変数の設定が必要です。この設定は、コンピュータ初心者には比較的難しい作業です。特に、会社貸与のコン

ピュータの場合などは、すでにインストール済みのソフトウェア製品によって環境変数が設定されていることがあります。この既存の設定を壊さないように、設定を追加する必要があります。

対応 Javaプログラミングに慣れることが目的なら、煩雑なセットアップなしでJava開発を体験できるdokojava（http://dokojava.jp）をぜひご利用ください。また、JDKの導入と環境変数の設定を行う場合、http://dokojava.jp/jdkにアクセスし、表示されるガイドに従って作業を進めれば、安全に設定できます。

参照 dokojavaのWebサイト

コンパイルができない（初級編）

C.3.1 「javac」と入力しても動かない

症状 コマンドラインプロンプトで「javac」と入力しても、『'javac' は、内部コマンドまたは外部コマンド、操作可能なプログラムまたはバッチ ファイルとして認識されていません。』というエラーメッセージが表示されます。

原因 JDKが正しくセットアップされていません。特にJDKインストール後に必要なPATH環境変数の設定がなされていない、または間違っている可能性があります。

対応 JDKのセットアップ手順を再度確認し、PATH環境変数が一字一句間違いなく正しく設定されているか確認します。それでも解決しない場合、一度JDKをアンインストールしてJDKセットアップを最初からやり直してみましょう。

参照 dokojavaのWebサイト

C.3.2 「java Main.java」と入力してコンパイルに失敗する

症状 コマンドラインプロンプトで「java Main.java」などと入力していますが、コンパイルできずに『Exception in thread "main" java.lang.NoClassDefFoundError: Main』というエラーメッセージが表示されます。

原因 コンパイルに使うコマンドを間違えています。javaコマンドではコンパイルはできません。

対応 javacコマンドを利用してください。

参照 付録A.3.3項

C.3.3 「javac Main」のような入力をして失敗する

症状 コマンドラインプロンプトで「javac Main」などと入力していますが、『クラス名 'Main' が受け入れられるのは、注釈処理が明示的に要求された場合だけです』というエラーメッセージが表示されます。

原因 コンパイル時に指定するファイル名を間違えています。

対応 ファイル名は拡張子(.java)まで含めて指定してください。「javac Main.java」が正しい指定方法です。

参照 付録A.3.3項目

C.3.4 『';'がありません』

症状 コンパイルすると、『';' がありません』というエラーメッセージが表示されます。しかし、その行には;を記述しています。

原因 ;(セミコロン)と似ている別の文字を入力している可能性があります。

対応 :(コロン)やi(小文字のアイ)を入力していないか確認します。また、誤って全角文字のセミコロンを入力していないか確認してください。

参照 1.2.2項

C.3.5 『構文解析中にファイルの終わりに移りました』

症状 コンパイルすると、『構文解析中にファイルの終わりに移りました』というエラーメッセージが表示されます。

原因 ソースコード内でブロックの波カッコ({…})の対応がとれていません(閉じ忘れ)。

対応 波カッコの対応を再度確認して修正します。なお、このエラーを未然に防ぐために、ブロックを開いたらすぐ閉じることや、正確なインデントを入力する習慣を付けましょう。

参照 1.2.2項、1.2.3項

C.3.6 『class、interface、または enum がありません。』

症状 コンパイルすると、『class、interface、または enum がありません。』というエラーメッセージが表示されます。

原因 ソースコード内でブロックの波カッコ({…})の対応がとれていません(閉じすぎ)。

対応 C.3.5と同じです。

参照 1.2.2項

C.3.7 『<identifier> がありません。』

症状 コンパイルすると、『<identifier> がありません。』というエラーメッセージが表示されます。

原因 ソースコード内でpublicやstaticなどキーワードのタイプミスや、現在のバージョンのJDKでは利用できないキーワード・構文を利用している場合に発生することがあります。

対応 エラーが指摘されている箇所でタイプミスをしていないか確認します。もし問題ない場合、現在利用中のJDKで利用できる構文であることを確認します。

C.3.8 「¥12288は不正な文字です。」

症状 コンパイルすると、『¥12288は不正な文字です。』というエラーメッセージが表示されます。

原因 ソースコード内に全角スペース文字が含まれています。たとえば、インデントにタブや半角スペースではなく全角スペースが含まれているなど、目視できないため気づきにくい間違いです。なお、Webサイトなどからサンプルコードをコピー&ペーストした場合に混入することもあるため注意が必要です。

全角スペース以外にも誤って混入しやすい全角文字には、¥65307の；、¥65373の｝、¥65289の）、¥8221の”があります。

対応 エラーが指摘されている行に全角スペースが含まれていないか、エディタで確認します。

参照 1.2.2項

C.3.9 「シンボルを見つけられません。」「～を～に解決できません」

症状 コンパイルすると、『シンボルを見つけられません。』『～を～に解決できません』というエラーメッセージが表示されます。

原因 存在しないクラス名・型名・変数名・メソッド名・フィールド名などを利用しようとしています。たとえば、String型を指定すべきところにstringとタイプミスした場合や、変数nameを使うつもりでnamaeを指定した場合に表示されるエラーです。

また importしていないクラスを利用しようとした際にも発生します。たとえば、java.util.ArrayListをインポートしていないにもかかわらず、コード中に「ArrayList a = new ArrayList();」という表記があると、クラスが見つからないとしてこのエラーが表示されます。

対応 このエラーメッセージの次の行には、見つからなかったシンボルが何であるかという情報が表示されています（たとえば、『シンボル：クラス string』）。

このシンボル名から、ソースコード中で確かに存在するクラス・型・フィールド・メソッド・変数を指定しているか確認します。もしクラスの場合、importを忘れていないことを確認します。

参照 1.2.2項、6.4.3項

C.3.10 「変数～は初期化されていない可能性があります。」（その1）

症状 コンパイルすると、『変数～は初期化されていない可能性があります。』というエラーメッセージが表示されます。

原因 値をまだ一度も代入していない変数を、計算や画面出力に利用しようとしています。

たとえば、「int a; System.out.println(a);」のようなプログラムは、aの内容が未定のまま利用されるためこのエラーの原因となります。

「メソッド開始直後に変数宣言だけを済ませ、利用する前に値を代入する」つもりが、実際には代入し忘れてこのエラーを引き起こすこともあります。

対応 エラーで報告された変数に、事前に値が代入されるようプログラムを修正します。

このエラーを未然に防ぐために、変数は極力必要になったときに随時宣言し、宣言と同時に初期値を代入するようにしましょう。

参照 4.2.3節

C.3.11 『変数〜は初期化されていない可能性があります。』(その2)

症状 final修飾されたフィールドを持つクラスをコンパイルすると、『変数〜は初期化されていない可能性があります。』というエラーメッセージが表示されます。

原因 final修飾されたフィールドは、インスタンス化が完了するまでの間に値が確定しなければなりません。このエラーは、コンストラクタの動作が終了した段階で、まだ何も代入されていないfinalフィールドが存在しうる場合に発生します。

たとえば、「final int a;」というフィールド宣言をしながら、コンストラクタ内でaに値を代入していない場合、このエラーの原因となります。

また、コンストラクタ内での条件分岐や例外処理によって、aへの代入が行われない可能性が少しでもあれば、このエラーが発生します。

対応 finalフィールド宣言時に「final int a = 10;」のように初期値を代入するか、コンストラクタ内でaに値が確実に代入されるようにします。

参照 1.3.5項

C.3.12 mainメソッドから、他のメソッドを呼び出せない

症状 mainメソッドから他のメソッドを呼び出すプログラムをコンパイルすると、『staticでないメソッド〜をstaticコンテキストから参照することはできません。』や『非staticメソッド〜をstatic参照できません。』というエラーメッセージが表示されます。

原因 mainメソッドはstaticであるため、静的メソッドの制約により、静的メンバ以外を利用することはできません。mainメソッドからstaticではないメソッドを呼び出そうとすると、このエラーの原因となります。

対応 mainメソッドから呼び出そうとしているメソッドがstaticで修飾されているか確認します。もしされていなければメソッドをstaticで修飾するか、呼び出し先メソッドを含むクラスをインスタンス化した上でメソッドを呼び出すようにします。

参照 9.3.3項

C.3.13 「クラス～は public であり、ファイル～で宣言しなければなりません。」

症状 プログラムをコンパイルすると、『クラス～は public であり、ファイル～で宣言しなければなりません。』や『public 型～はそれ独自のファイル内に定義されなければなりません。』というエラーメッセージが表示されます。

原因 public なクラス A を定義するファイルの名前は、A.java である必要があります。それ以外のファイル名にしている場合、このエラーの原因となります。

対応 「クラス名」と「ファイル名から拡張子を取り除いたもの」が一致するように、クラス名かファイル名のいずれかを修正します。

参照 1.2.1 項

コンパイルができない（中級編）

C.4.1 「return 文が指定されていません。」「このメソッドは型～の結果を返す必要があります。」

症状 プログラムをコンパイルすると、『return 文が指定されていません。』や『このメソッドは型～の結果を返す必要があります。』というエラーメッセージが表示されます。

原因 戻り値を返すように宣言されたメソッドの内部で、「戻り値を返さない可能性が少しでもある場合」にこのエラーが発生します。

たとえば、int 型を返すメソッドの中身が「if(条件式) { return 1; }」だけの場合、条件式が満たされない場合には値が戻されないためエラーとなります。また、処理中に例外が発生した場合に return が実行されない可能性があるためこのエラーが指摘されることもあります。

対応 処理中の条件分岐や例外発生の有無によらず、常に何らかの値を return するようにプログラムを修正します。

参照 5.3 節

C.4.2 「～は～で private アクセスされます」「～は不可視です」

症状 プログラムをコンパイルすると、『～は～で private アクセスされます』や『～は不可視です』というエラーメッセージが表示されます。

原因 カプセル化によりアクセス権限がないクラス・フィールド・メソッドを利用しようとしています。たとえば、private 宣言されているフィールドを他のクラスから利用しようとした場合に、このエラーの原因となります。

対応 利用しようとしているクラスやメンバのアクセス修飾が正しいことを確認します。private フィールドへのアクセスが必要な場合、フィールドのアクセス修飾を public などへ緩めるのではなく、getter や setter がすでに存在しないか確認し、なければ作成を検討します。

参照 10.2.2 項、10.3 節

C.4.3 コンストラクタを定義しても new で利用できない

症状 新しいコンストラクタを定義したにもかかわらず、new 演算子を用いてインスタンス化する際に利用できません。たとえば、Hero クラスに引数 2 つのコンストラクタを「public void Hero(String name, int hp)」と宣言しても、「new Hero("ミナト", 100);」とするとエラーが発生します。

原因 コンストラクタの宣言に間違いがあり、コンストラクタとして見なされていない可能性があります。よくある間違いは、「戻り値を宣言している」「クラス名とコンストラクタ名に違いがある」というものです。

対応 コンストラクタの宣言に戻り値を付けていないことを確認します（void の記述もダメです）。次に、コンストラクタ名がクラス名と完全に一致していることを確認します。

参照 9.2.3 項

C.4.4 オーバーロードできない

症状 既存のメソッド「public void m(int a)」が存在するプログラムにおいて、戻り値が違うメソッド「public int m(int a)」を定義してオーバーロードを行おうとしたところ、『～は～で定義されています。』というエラーが表示されます。

原因 「戻り値だけしか違いがない」場合には、オーバーロードは利用できません。

対応 メソッド名を変更してオーバーロードを諦めるか、引数の数か型が違うものになるようにします。もし既存メソッドと新メソッドの戻り値に継承関係や共通の親クラスが存在するならば、親クラスの型を戻り値とすることでメソッドを統一できます。

参照 5.4 節

C.4.5 メソッド呼び出しの戻り値を Date 型変数に代入できない

症状 クラス A の a() メソッドは、Date 型の戻り値を返します。クラス B の b() メソッドの内部で a() を呼び出し、その結果を Date 型変数に代入しようとすると、『互換性のない型』や『型の不一致』というコンパイルエラーが発生します。

原因 java.util.Date 型で返されたインスタンスを、java.sql.Date 型の変数に代入しようとしている可能性があります。上記の例では A.java で java.util.Date が、B.java で java.sql.Date が import されている場合にこのエラーが発生します。

特に、統合開発環境の入力支援機能で java.sql.Date の import 文を誤って追加してしまったことに開発者が気づかず、このエラーの原因となることがあるため注意が必要です。

対応 正しい import 文が記述されているかソースコードを確認します。なお、java.util.Date クラス以外にも入力支援により import 間違いしやすいクラスとしては、java.util.List と java.awt.List があります。

参照 15.1.1 節

C.4.6 「例外～は報告されません。」「処理されない例外の型～」

症状 プログラムをコンパイルすると、『例外～は報告されません。』『処理されない例外の型～』というエラーが表示されます。

原因 あるメソッドAの中で「チェック例外が発生する可能性のあるメソッドB」を呼び出していますが、例外発生時にどのように動作をすべきかメソッドAに指定していません。具体的には、try-catch と throws のどちらも指定されていません。

対応 もし例外発生時にメソッドA内で何らかの対処を行う場合、メソッドB呼び出しを try-catch 文で囲みます。 例外発生時にメソッドA内では対処を行わず、その呼び出し元に例外処理を委譲する場合は、メソッド修飾に throws を追加します。

参照 15.2.2 項、15.6.3 項

C.4.7 「シンボルを見つけられません。」(その 2)

症状 プログラムをコンパイルすると、『シンボルを見つけられません。』や『暗黙的スーパー・コンストラクター ～ は、デフォルト・コンストラクターについては未定義です。』というエラーが表示されます。

原因 親クラスAに引数があるコンストラクタだけが宣言されていて、その子クラスBにコンストラクタが宣言されていない場合に発生するエラーです。

　子クラスBに暗黙的に追加されるデフォルトコンストラクタは、親クラスAの引数なしコンストラクタを呼び出そうとしますが、親クラスに引数なしコンストラクタがないためにエラーとして報告されます。

対応 子クラスに明示的にコンストラクタを記述し、その中で super() を用いて親コンストラクタに引き渡す引数を指定します。

参照 11.3 節

プログラムが実行できない

C.5.1 『java.lang.NoClassDefFoundError』(その 1)

症状 プログラムを「java Main.java」のように実行すると複数行のエラーが表示されます。1行目には『java.lang.NoClassDefFoundError: ～/java』というエラーメッセージが表示されます。

原因 プログラム起動時に拡張子(.java)を含むファイル名を指定しています。起動時に指定すべきは、起動したいクラスの完全限定クラス名です。

対応 「java Main」のような記述でjavaコマンドを起動してください。なお、Main クラスがパッ

ケージに所属している場合は、完全限定クラス名を指定してください。

参照 付録A.3.4項

C.5.2 『java.lang.NoClassDefFoundError』（その2）

症状 プログラムを「java Main」のように実行すると複数行のエラーが表示されます。1行目には『java.lang.NoClassDefFoundError: ～』というエラーメッセージが表示されます。

原因 何らかの理由で、JVMが指定されたクラスのクラスファイルを見つけられません。具体的な理由としては、次のものが考えられます。

①起動クラスのクラス名のつづりをタイプミスしている。

②起動クラスを完全限定クラス名（パッケージ名付きのクラス名）で指定していない。

③クラスファイルがクラスパス以下のパッケージ階層に対応したフォルダに配置されていない。

④クラスパスの指定が正しく行われていない。

対応 起動しようとしているクラス名をタイプミスしていないか確認します。たとえば、クラス名がMain（先頭が大文字）でありながら「java main」のように小文字で起動していないかをです。

次に、起動しようとしているクラスの完全限定クラス名を確認します。パッケージa.b.cのMainクラスを起動するならば、コマンドラインからは「java a.b.c.Main」と入力すべきです。

さらに、クラスファイルが所定の場所に正しく存在するか確認します。特にパッケージに所属するクラスではファイル置き場にも注意が必要です。上記のMainクラスであれば、「（クラスパスのフォルダ）¥a¥b¥c¥Main.class」にクラスファイルが存在していなければなりません。

最後に、クラスパスが起動時または環境変数で正しく設定されているかを確認します。タイプミスをしていないか注意してください。

参照 6.7節

C.5.3 『java.lang.NoSuchMethodError: main』

症状 プログラムを実行すると複数行のエラーが表示されます。1行目には『java.lang.NoSuchMethodError: main』というエラーメッセージが表示されます。

原因 起動しようとしたクラスに、正しいmainメソッドがありません。mainメソッドを定義したつもりなのにこのメッセージが表示される場合、細部が間違っている（publicやstaticを付け忘れている、引数の型が違う、mainのつづりが違う）可能性があります。

対応 起動しようとしたクラスにmainメソッドが存在することを確認します。特にmainメソッドの宣言は一字一句間違えていないことを確認します。

参照 1.2.2項

C.5.4 『java.lang.ArrayIndexOutOfBoundsException』

症状 プログラムを実行すると複数行のエラーが表示されます。1行目には『java.lang.ArrayIndexOutOfBoundsException: ～』というエラーメッセージが表示されます。

原因 ある配列について、範囲外の添え字を使ってアクセスしようとしました。たとえば、要素数3の配列iを確保している場合に、i[3]やi[-1]をアクセスしようとするとこのエラーが発生します。

対応 エラーが報告されている行で、添え字が範囲外でないか確認します。特にforやwhileによるループの中で配列を用いている場合、ループの回数に注意します。なお、このエラーメッセージの末尾にあるコロンの右側には、「何番目の要素にアクセスしようとしたか」を示す数字が表示されていますので、参考にしてください。ちなみにループにおけるこのエラーを未然に防ぐためには、拡張for文の利用が有効です。

参照 4.3.1項

C.5.5 『java.lang.NullPointerException』

症状 プログラムを実行すると複数行のエラーが表示されます。1行目には『java.lang.NullPointerException』というエラーメッセージが表示されます。

原因 null(が格納されている変数)に対して、そのフィールドやメソッドを利用しようとしました。たとえば、変数sがnullであるときに「s.toString();」を実行すると、このエラーが発生します。

また、「a.method1().method2()」のようにメソッドの呼び出しを連ねて記述している場合に「a.method1()」の実行結果としてnullが返ってきているためにmethod2()の呼び出しでエラーが発生している可能性があります。

対応 エラーが報告されている行の中で、メソッドやフィールドを利用している変数(xxx.yyyの場合のxxx部分)を特定し、内容がnullでないか確認します(エラーになる行の直前で変数の内容を画面に出力して確認できます)。

もしnullの場合、ソースコードを逆に遡って、「どこでnullになってしまったのか」を特定し、その箇所を訂正します。

参照 4.6.3項

C.5.6 『java.lang.ClassCastException』

症状 プログラムを実行すると複数行のエラーが表示されます。1行目には『java.lang.ClassCastException: ～ cannot be cast to ～』というエラーメッセージが表示されます。

原因 あるインスタンスを、そのインスタンスの型または親クラスの型以外へキャストしようとした場合に、このエラーが発生します。たとえば、Heroインスタンスが入っている

Object 型変数 o がある場合、「Wizard w = (Wizard)o;」のようなコードを記述すると、Hero インスタンスを Wizard と見なそうと試みて失敗します。

多くの場合、キャストを試みた変数（上記の例では o）に、実際には何型のインスタンスが入っているかを開発者が理解していないことに起因しています。

対応 キャストを試みた変数（上記の例では o）にインスタンスを代入している箇所までソースコードを遡ります。そして、変数に代入したインスタンスの型（何型として new されたインスタンスを代入しているか）を確認します。次に、キャストの変換先の型（上記例では Wizard）を確認します。変換先の型がインスタンスの型と同じか、その親クラスの型でなければキャストを行うことはできません。

参照 13.4.2 項

C.5.7 『java.lang.ArithmeticException: / by zero』

症状 プログラムを実行すると複数行のエラーが表示されます。1 行目には『java.lang.ArithmeticException: / by zero』というエラーメッセージが表示されます。

原因 数学的に許されていない「0 での割り算」を行いました。たとえば、「int a = b / c;」のようなコードがあるときに、c に 0 が入っていた場合に発生します。特に、ユーザーが入力した情報やファイルから読み込んだ情報を割る数（上記の例の c）として利用した場合に、入力値チェックをしていないと、このエラーの原因となります。

対応 どのような条件での動作でも 0 で除算が行われないようプログラムを修正します。

参照 15.1.2 項

エラーは出ないが動作がおかしい

C.6.1 プログラムが動き続けて終わらない

症状 プログラムを実行すると、動き続けて止まりません。

原因 ①プログラムはユーザーからのキーボード入力を待っている状態です。2 章で登場した java.util.Scanner などを使ってキー入力を試みるとこの状態になります。

②プログラムは無限ループに陥っています。誤った終了条件の for 文などを記述すると、いつまでも処理が繰り返されます。

対応 コマンドラインプロンプトの場合、CTRL キーを押しながら C キーを押すと実行中のプログラムを強制終了できます。Eclipse や NetBeans などの統合開発環境の場合、操作メニューからプログラムを終了させます。

①キーボード入力待ちによってプログラムが終わらないことは正しい動作です。ただし利用者に対して「キーボード入力を待っている」ことが伝わるように、「○○を入力してください > 」のようなプロンプトを出すようにします。

②無限ループはプログラミングミスですのでループの終了条件を確認し、ループの中でカウンタ変数を次に進めているかを確認します。

参照 2.6.6 節、3.6.3 節

C.6.2 if による条件分岐が正しく動作しない

症状 「if(str == "hello") {…} 」のような条件分岐を含むプログラムを実行すると、エラーは出ませんが、str に "hello" が入っているにも関わらずブロック内が実行されません。

原因 文字列の内容の比較に == を使っています。

対応 文字列の内容が等しいかどうかの判定には、== 演算子ではなく equals() メソッドを利用します。

参照 3.3.3 項

C.6.3 金額やフォルダ名の画面表示がおかしい

症状 「System.out.println("¥1200");」や「System.out.println("C:¥notepc");」のように記述したプログラムを実行すると、エラーは出ませんが、画面におかしな文字が表示されたり、表示が崩れます。

原因 ¥ 記号は文字の Unicode 表記やエスケープシーケンスとして利用される特殊文字です。「¥1200」の場合、「¥120」が文字「P」に置き換わります。文字列リテラル中で ¥ 文字を用いたい場合には「¥¥」という表記が必要です。

対応 「¥」の代わりに「¥¥」を利用します。たとえば、「System.out.println("¥¥1200");」とします。

参照 2.2.2 項

C.3 エラーメッセージ別索引

エラーメッセージ	解説番号
～は～で private アクセスされます 。	C.4.2
～は～で定義されています。	C.4.4
～は不可視です。	C.4.2
～を～に解決できません。	C.3.9
¥12288 は不正な文字です。	C.3.8
';' がありません 。	C.3.4
class、interface、または enum がありません。	C.3.6
<identifier> がありません。	C.3.7
java.lang.ArithmeticException: / by zero	C.5.7
java.lang.ArrayIndexOutOfBoundsException	C.5.4
java.lang.ClassCastException	C.5.6
java.lang.NoClassDefFoundError: ～	C.3.2、C.5.1、C.5.2
java.lang.NoSuchMethodError: main	C.5.3
java.lang.NullPointerException	C.5.5
'javac' は、内部コマンドまたは外部コマンド、操作可能なプログラムまたはバッチ ファイルとして認識されていません。	C.3.1
public 型～はそれ独自のファイル内に定義されなければなりません 。	C.3.13
return 文が指定されていません。	C.4.1
static でないメソッド～を static コンテキストから参照することはできません 。	C.3.12
暗黙的スーパー・コンストラクタ ～ は、デフォルト・コンストラクタについては未定義です。	C.4.7
型の不一致。	C.4.5
クラス～は public であり、ファイル～で宣言しなければなりません。	C.3.13
クラス名 'Main' が受け入れられるのは、注釈処理が明示的に要求された場合だけです 。	C.3.3
構文解析中にファイルの終わりに移りました。	C.3.5
互換性のない型。	C.4.5
このメソッドは型～の結果を返す必要があります。	C.4.1
処理されない例外の型～ 。	C.4.6
シンボルを見つけられません。	C.3.9、C.4.7
非 static メソッド～を static 参照できません 。	C.3.12
変数～は初期化されていない可能性があります。	C.3.10、C.3.11
例外～は報告されません。	C.4.6

付録 D

構文リファレンス

ここでは、本書で扱ったさまざまな Java の構文をまとめて掲載します。各構文の詳細については、それぞれの構文に記載した 参照 ページを参照してください。
なお、このリファレンスは実用上よく用いる構文のみをわかりやすく紹介することを目的としており、厳密な言語仕様とは異なる部分も含まれます。正確な構文規則を知る必要がある場合は、Web で公開されている「Java 言語仕様」を確認してください。

凡例

[= 初期値]……………「= 初期値」部分は省略可能

文………………………「文」を繰り返し記述

引数 ,… ………………「引数」をカンマ区切りで繰り返し記述

(abstract | final)………abstract か final のどちらかを記述

[abstract | final]………abstract か final のどちらかを記述するか、何も記述しない

⇒クラス宣言 …………この付録の クラス宣言 を参照

Java ソースファイル

```
[ package パッケージ名 ; ] ········· 参照 p.231
[ ⇒インポート宣言 ] …
[ ⇒クラス宣言 | ⇒インタフェース宣言 ] …
```

インポート宣言

```
      import パッケージ名 .*; ········· 参照 p.241
または import パッケージ名 . クラス名 ; ········· 参照 p.241
```

クラス宣言

```
[ public ] ········· 参照 p.394
[ abstract | final ] ········· 参照 p.418、p.464
class クラス名 ········· 参照 p.308
[ extends 親クラス名 ] ········· 参照 p.412
[ implements 親インタフェース名 , … ] { ········· 参照 p.480
        [ ⇒メンバ宣言 ] …
}
```

インタフェース宣言

```
[ public ] ········· 参照 p.394
interface インタフェース名 ········· 参照 p.476
[ extends 親インタフェース名 , … ] { ········· 参照 p.486
        [ ⇒抽象メソッド宣言 ] …
}
```

メンバ宣言

```
      ( ⇒フィールド宣言 | ⇒コンストラクタ宣言 | ⇒メソッド宣言 )
または ⇒抽象メソッド宣言
```

フィールド宣言

```
[ public | protected | private ] ········· 参照 p.380
[ static ] [ final ] ········· 参照 p.309、p.363
型名　フィールド名 ········· 参照 p.309
[ = 初期値 ] ; ········· 参照 p.309
```

コンストラクタ宣言

```
[ public | protected | private ] ········· 参照 p.380
クラス名 ( [ 引数 , … ] ) ········· 参照 p.352
[ throws 例外クラス名 , … ] { ········· 参照 p.589
        [ ⇒コンストラクタ呼び出し ]
        [ ⇒文 ] …
}
```

コンストラクタ呼び出し

```
      this( [ 引数 , … ] ) ········· 参照 p.360
または super( [ 引数 , … ] ) ········· 参照 p.429
```

メソッド宣言

```
[ public | protected | private ] ········· 参照 p.380
[ static ] [ final ] ········· 参照 p.367、p.419
戻り値　メソッド名 ( [ 引数 , … ] ) ········· 参照 p.182
[ throws 例外クラス名 , … ] { ········· 参照 p.589
        [ ⇒文 ] …
}
```

抽象メソッド宣言

```
[ public | protected ] ········· 参照 p.380
abstract ········· 参照 p.463
戻り値　メソッド名 ( [ 引数 , … ] ) ········· 参照 p.182
[ throws 例外クラス名 , … ] ; ········· 参照 p.589
```

文

```
      ( ⇒制御構文 | ⇒変数宣言の文 | ⇒式の文 )
または ( break; | continue; ) ········· 参照 p.128
または return [ 戻り値 ]; ········· 参照 p.186
または throw 例外インスタンス ; ········· 参照 p.592
または ⇒ try-catch 文
```

制御構文

```
      ( ⇒ if 文 | ⇒ switch 文 )
または ( ⇒ for 文 | ⇒ while 文 | ⇒ do-while 文 )
```

if 文

```
if ( 条件式 ) { ········· 参照 p.114
        [ ⇒文 ] …
```

```
} [ else if ( 条件式 ) {
        [ ⇒文 ] …
} ]… [ else {
        [ ⇒文 ] …
} ]
```

switch 文

```
switch ( 条件値 ) { ···························· 参照 p.117
        [ case 値 :
                [ ⇒文 ] …
                [ break; ]
        ]…
        [ default :
                [ ⇒文 ] …
] }
```

for 文

```
for( 初期処理 ; ループ継続条件式 ; 周回毎処理 ) { ········ 参照 p.124
        [ ⇒文 ] …
}
```

while 文

```
while ( ループ継続条件式 ) { ························· 参照 p.121
    [ ⇒文 ] …
}
```

do-while 文

```
do { ··············································· 参照 p.121
        [ ⇒文 ] …
} while( ループ継続条件式 );
```

try-catch 文

```
try [ ( [⇒文]…) ] { ································ 参照 p.580、p.586
        [ ⇒文 ] …
} [ catch( 例外クラス名 |… 変数名 ) { ················· 参照 p.580
        [ ⇒文 ] …
} ]… [ finally { ···································· 参照 p.584
        [ ⇒文 ] …
} ]
```

変数宣言の文

```
[ final ] ·········································· 参照 p.053
型名　変数名 ········································ 参照 p.045
[ = 初期値 ]; ······································· 参照 p.051
```

式の文

```
⇒式 ;
```

式

```
        ( 値 | 変数名 )
または  式 [ 演算子 式 ] ································ 参照 p.061
または  ⇒インスタンス生成式
または  ( ⇒メンバアクセス式 | ⇒静的メンバアクセス式 )
または  ⇒配列生成式
または  ⇒配列アクセス式
または  this ········································ 参照 p.311
または  super ······································· 参照 p.425
```

インスタンス生成式

```
new クラス名 ( [ 引数 , … ] ) ························ 参照 p.318
```

メンバアクセス式

```
        [ クラス型変数名 .] フィールド名 ·················· 参照 p.319
または  [ クラス型変数名 .] メソッド名 ( [ 引数 , …] ) ····· 参照 p.319
```

静的メンバアクセス式

```
        [ [ パッケージ名 .] クラス名 .] フィールド名 ·········· 参照 p.364
または  [ [ パッケージ名 .] クラス名 .] メソッド名 ( [ 引数 , … ] ) 参照 p.367
```

配列生成式

```
        new 要素型名 [ 要素数 ] ··························· 参照 p.143
または  new 要素型名 [] { [ 値 , … ] } ····················· 参照 p.147
または  { [ 値 , … ] } ···································· 参照 p.147
```

配列アクセス式

```
配列型変数名 [ 添え字 ] ····································· 参照 p.145
```

INDEX

記号

数字

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Z

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

な

ぬ

ね

は

ひ

ふ

へ

ほ

ま

む

め

も

ゆ

よ

ら

り

る

れ

ろ

わ

■著者

中山清喬（なかやま・きよたか）

株式会社フレアリンク代表取締役。IBM 内の先進技術部隊に所属しシステム構築現場を数多く支援。退職後も研究開発・技術適用支援・教育研修・執筆講演・コンサルティング等を通じ、「技術を味方につける経営」を支援。現役プログラマ。講義スタイルは「ふんわりスパルタ」。

国本大悟（くにもと・だいご）

文学部・史学科卒。大学では漢文を読みつつ、IT 系技術を独学。会社でシステム開発やネットワーク・サーバ構築等に携わった後、フリーランスとして独立する。システムの提案、設計から開発を行う一方、プログラミングやネットワーク等の IT 研修に力を入れており、大規模 SIer やインフラ系企業での実績多数。

■イラスト

高田ゲンキ（たかた・げんき）

イラストレーター／神奈川県出身／ 1976 年生。東海大学文学部卒業後、デザイナー職を経て、2004 年よりフリーランス・イラストレーターとして活動。書籍・雑誌・Web・広告等で活動中。
ホームページ　http://www.genki119.com

STAFF
編集　石塚康世
　　　片元 諭
イラスト　高田ゲンキ
DTP 制作　SeaGrape ／佐藤 卓／眞部謙二
カバーデザイン　阿部 修（G-Co.Inc.）
カバー制作　高橋結花
編集長　玉巻秀雄

●本書の内容に関するご質問は、書名・ISBN（このページの下に記載）・お名前・電話番号と、該当するページや具体的な質問内容、お使いの動作環境などを明記のうえ、インプレスカスタマーセンターまでメールまたは封書にてお問い合わせください。電話やFAX等でのご質問には対応しておりません。なお、本書の範囲を超える質問に関しましてはお答えできませんのでご了承ください。
●落丁・乱丁本はお手数ですがインプレスカスタマーセンターまでお送りください。送料弊社負担にてお取り替えさせていただきます。但し、古書店で購入されたものについてはお取り替えできません。

■読者の窓口
インプレスカスタマーセンター
〒101-0051　東京都千代田区神田神保町一丁目105番地
電話　03-6837-5016／FAX　03-6837-5023
info@impress.co.jp

■書店／販売店のご注文窓口
株式会社インプレス 受注センター
TEL　048-449-8040
FAX　048-449-8041

スッキリわかるJava入門 第2版

2014年　8月11日　初版発行
2016年　3月　1日　第1版第6刷発行

著　者　中山清喬／国本大悟

発行人　土田米一

発行所　株式会社インプレス
〒101-0051　東京都千代田区神田神保町一丁目105番地
TEL 03-6837-4635（出版営業統括部）
ホームページ http://book.impress.co.jp/

印刷所　日経印刷株式会社

ISBN978-4-8443-3638-9　C3055
Printed in Japan